NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AR PARAMETER ESTIMATION
USING TMS320C30
DIGITAL SIGNAL PROCESSOR CHIP

by

Mücahit Karasu

December, 1995

Thesis Advisor : Michael K.Shields
Co-Advisor : Murali Tummala

Approved for public release; distribution is unlimited.

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1995	Master's Thesis

4. TITLE AND SUBTITLE AR PARAMETER ESTIMATION USING TMS320C30 DIGITAL SIGNAL PROCESSOR CHIP	5. FUNDING NUMBERS
6. AUTHOR(S) Mücahit Karasu	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited.	

13. ABSTRACT *(maximum 200 words)*
Autoregressive analysis is used in modern signal processing applications for modeling and estimation of random signals. High speed digital signal processors with advanced architecture and special digital signal processing instructions, mostly compiled in C language, can be used in these applications to achieve realtime performance. A commercially available digital signal processor has been used in this work to estimate the AR parameters and power spectral density from the given input data by using the Levinson, Burg and Schur algorithms. This work produced a library file that contains the object files of the AR parameter estimation algorithms. The time required in terms of the cycle counts to execute each algorithm is listed for different data lengths and model orders.

14. SUBJECT TERMS Digital Signal Processing, AR algorithms, DSP chips, Realtime implementation	15. NUMBER OF PAGES 110
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited.

AR PARAMETER ESTIMATION USING TMS320C30 DIGITAL SIGNAL PROCESSOR CHIP

Mücahit Karasu
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1989

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1995

Author: _____
Mücahit Karasu

Approved by: _____
Michael K.Shields, Thesis Advisor

_____
Murali Tummala, Co-Advisor

_____
Herschel H. Loomis, Jr., Chairman
Department of Electrical and Computer Engineering

ABSTRACT

Autoregressive analysis is used in modern signal processing applications for modeling and estimation of random signals. High speed digital signal processors with advanced architecture and special digital signal processing instructions, mostly compiled in C language, can be used in these applications to achieve realtime performance. A commercially available digital signal processor has been used in this work to estimate the AR parameters and power spectral density from the given input data by using the Levinson, Burg and Schur algorithms. This work produced a library file that contains the object files of the AR parameter estimation algorithms. The time required in terms of the cycle counts to execute each algorithm is listed for different data lengths and model orders.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to thank all the instructors, lab technicians and curriculum staff in Electrical Engineering Department for their contribution toward my education at the Naval Postgraduate School.

I would like to thank my thesis advisor Professor Mike Shields for his support and assistance during my study. I would also like to thank Professor Murali Tummala for being my co-advisor.

Most importantly, I would like to thank my parents who were miles away for 30 months and kept on praying for me. They always provided the best advice, enduring faith and continuous support. Mom and Dad, I love you very much.

I. INTRODUCTION

Almost every field of science and engineering, such as acoustics, physics, telecommunications, data communications, control systems and radar, deal with signals. Digital Signal Processing (DSP) is concerned with the representation of signals using numerical techniques and digital processors. Several DSP micro processors have become commercially available in Very Large Scale Integration (VLSI) form. These devices, such as Texas Instruments TMS320, Analog Devices ADSP2100 and Motorola DSP56000 family of digital signal processors (however, this is not an endorsement of any of these products by the U.S. Government) are high speed micro processors, designed specifically to perform DSP algorithms. By taking advantage of their advanced architectures, parallel processing capabilities, and special DSP instructions, these devices can execute millions of DSP operations per second. One may find different kinds of DSP chips in the market today with different characteristics, including speed, cost and on-chip memory tailored to application areas.

Autoregressive (AR) analysis and Linear Prediction (LP) are used in modern signal processing applications for modeling and estimation with random signals. The Linear Prediction is based on estimating the current sample of a given input signal using the linear combinations of its past samples. By minimizing the power of the error signal between the actual signal samples and the predicted ones, a set of predictor parameters can be found. There are many different methods of extracting reasonable estimates of the model parameters. Digital signal processors programmed in assembly or in C language can be used to perform various AR and LP algorithms in real time. In this study, we are mainly interested in implementation of the following algorithms for AR analysis on the TMS320C30 digital signal processor:

1. Levinson Algorithm,
2. Burg Algorithm,
3. Schur Algorithm,
4. AR Spectral Estimation.

The Levinson algorithm provides a computationally efficient way to obtain the prediction parameters of an AR process from its autocorrelation coefficients and can be used to find the appropriate model order. The Burg algorithm minimizes the sum of both forward and backward prediction error powers directly from the given input data without computing the autocorrelation values of the input to find the prediction parameters. The Schur algorithm is an alternative to the Levinson algorithm which takes advantage of gapped functions to find the prediction parameters from the autocorrelation function. The split Schur algorithm reduces the computation complexity 50% by computing only half of the multiplication terms. An estimate of the Power Spectral Density (PSD) of an AR process can be obtained by using the prediction coefficients and the prediction error variance from the AR algorithms. [Ref. 7-10]

This work produced a library of real time parameter estimation algorithms and demonstrated their use with the TMS320C30 digital signal processor. The library contains the object files of the AR routines mentioned above. These subroutines may be called from any C language program by linking the library.

In the next chapter, background information on the TMS320C30 digital signal processor's architecture is presented briefly. Chapter III describes the data transfer between the PC and the C30 board and the development tools for compiling the programs. The AR algorithms used in this work are presented in Chapter IV. The implementation of the algorithms in C language code using the C30 compiler utilities and the results in terms of the number of cycles to execute the AR routines are described in Chapter V. The conclusions and recommendations are discussed in Chapter VI. The C language code of the AR routines are listed in Appendix A, and the assembly language code for the autocorrelation function is listed in Appendix B. All the batch files used to obtain the object files, to create the library, and to execute the C30 and the PC programs are listed in Appendix C. An example data transfer program to check the results of the AR routines using the C30 board and the PC is described in Appendix D, and an interrupt service routine example is described in Appendix E. The memory map file LSICMAP.CMD for the C30 board is listed in Appendix F.

II. TMS320C30 (C30) ARCHITECTURE

TMS320 is a generic name for a family of digital signal processors manufactured by Texas Instruments (TI) and designed to support a wide range of high speed applications. The TI TMS320 family consists of six generations: TMS320C1x, TMS320C2x, TMS320C3x, TMS320C4x, TMS320C5x and TMS320C8x. Some specific features are added in each generation to provide different cost/performance tradeoffs.

The TMS320C3x generation has two different versions, the TMS320C30 and the TMS320C31. Each version is supplied with different rated clock speeds. The TMS320C30 is available with these clock speeds:

- TMS320C30: 60-ns single cycle execution time.
- TMS320C30-27: 74-ns single cycle execution time.
- TMS320C30-40: 50-ns single cycle execution time.

All the processors in the TMS320C3x generation have the same instruction set, similar peripherals, and different timing and electrical characteristics.

The TMS320C30 (C30) is a 32-bit digital signal processor with a 60-ns single cycle instruction execution time or 16.7 million instructions per second (MIPS). While an instruction is being executed, the next three instructions are being consequently fetched, decoded and read. The C30 can perform many instructions in parallel, such as multiply and add, in a single cycle producing a minimum instruction cycle time of 30-ns or 33.3 MIPS. Some of the important features of the C30 are:

- 4K x 32 bit words of on-chip ROM.
- 2K x 32 bit words of on-chip RAM.
- 64 x 32-bit instruction cache.
- 32-bit instruction and data words, 24-bit addresses.
- 29 32-bit registers used for addressing stack management, processor status, interrupts and block repeat.
- Integer, floating point and logical operations.
- Parallel ALU and multiplier instructions in a single cycle.

- Two and three operand instructions.
- On-chip direct memory access (DMA) controller for concurrent I/O and CPU operation.
- Conditional call and returns.
- Two 32-bit timers.
- Two serial ports to support 8/16/24/32-bit transfers.
- Two general-purpose external flags and four external interrupts.

The C30 can be programmed either in its special assembly language instructions or in C language and compiled by the C30 compiler. In many applications where execution time is critical, it may be necessary to write specific functions in assembly language.

The C30 exhibits some common characteristics of general purpose processors, such as a rich instruction set and hardware/software interrupts, which make it easy to use. Because of these features, C30 has been used in a number of real time applications including communications, control, speech, and image processing.

Some very basic information about the TMS320C30 architecture is presented in the following sections. More information and examples can be found in *TMS320C3x User's Guide* [Ref. 1], *Digital Signal Processing with C and TMS320C30* by Rulph Chassaing [Ref. 4], and the course notes of *EC 4930 - Digital Signal Processing Hardware* [Ref. 5]. The memory organization is addressed first, followed by the CPU and registers, addressing modes, instruction set, and peripherals.

A. MEMORY ORGANIZATION

There is a total of 16 million 32-bit words of memory space containing program, data and input/output. The addresses of memory locations shown in Figure 2-1 are from 0h to 0FFFFFFh. There are two RAM blocks of on chip memory each with 1K words. There is one ROM block of memory with 4K words. The C30 can be used in microcomputer or microprocessor modes. Both modes are similar, except that the ROM block is not used in microprocessor mode.

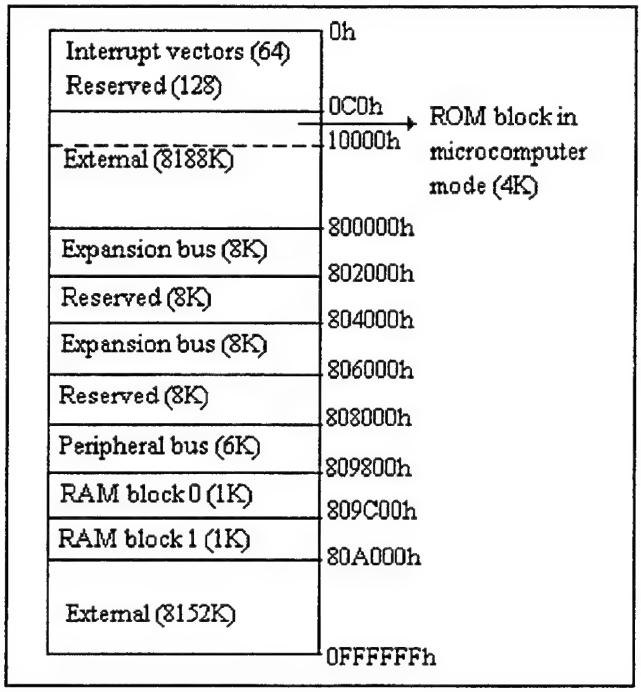

Figure 2-1. Memory Locations of TMS320C30

Memory addresses 0h through 0BFh are for hardware and software interrupt vectors locations and other reserved locations. Locations from 0Ch to 800000h are mapped to the off-chip bus. In microcomputer mode, memory space from 0Ch to 10000h is assigned to the ROM block instead of the off-chip memory. [Ref. 1]

Memory addresses from 800000h to 802000h and from 804000h to 806000h are mapped to expansion buses and used for external peripheral devices. The locations from 808000h to 809800h are mapped to on-chip peripherals. Two timers, a DMA controller, and two serial ports and their control registers are contained in this section of the memory. Memory locations from 809800h to 80A000h are mapped to RAM blocks. The locations from 802000h to 804000h and from 806000h to 808000h are reserved and reads or writes are not allowed here. Memory locations from 80A000h to 0FFFFFFh are mapped to the main off-chip bus and used for external memory. [Ref. 1]

B. CENTRAL PROCESSING UNIT AND REGISTERS

In order to execute the sophisticated DSP algorithms in real time, the C30 has a register based Central Processing Unit (CPU) architecture with nine buses associated with memory access. This allows the CPU to access independently each component of the memory. The CPU contains the Multiplier, Arithmetic Logic Unit (ALU), 32-bit Barrel Shifter, Internal Buses, Auxiliary Register Arithmetic Units and Register Files.

The *Multiplier* performs single cycle multiplications on integer and floating point data. Single cycle operations on integer, logical and floating point data, and conversions of single cycle integer and floating points are performed by the *ALU.* The *Auxiliary Register Arithmetic Units (ARAUs)* generates two addresses in a single cycle. They support addressing with displacements and index registers by operating in parallel with the multiplier and ALU. The *Barrel Shifter* is used to shift up to 32 bits left or right in a single cycle.

The *Internal Buses* connect the on-chip memory, off-chip memory, and on-chip peripherals. The separate program buses (PADDR and PDATA), data buses (DADDR1, DADDR2 and DDATA), and DMA buses (DMAADDR and DMADATA) allow for parallel program fetches, data accesses, and DMA accesses. These operations are performed by carrying two operands from memory and two operands from the register file at the same time.

There are 28 registers in the CPU register file. These registers can be operated by the multiplier and ALU and can be used for addressing, stack management, processor status, interrupts, and block repeat. The C30 register file contains the following registers:

1. *R0-R7, extended precision registers*, are used for storing and supporting 32-bit integer and 40-bit floating point numbers.

2. AR0-AR7, auxiliary registers, are mainly used for holding address values. They are directly connected to two Auxiliary Register Arithmetic Units (ARAU) to update the two address values in every instruction cycle. These registers can also be used as loop counters or as general purpose registers.

3. DP, data page pointer, are used in direct addressing as a pointer to the page of data being addressed. There are 256 data pages, each 64K words long.

4. IR0 and IR1, index registers, are used for indexing the address by the ARAUs.

5. *BK, block size register,* are used by the ARAUs to specify the data block size in circular addressing.

6. *SP, system stack pointer,* contains the address of the top of the stack. The SP always points to the last element pushed on to the stack, and is manipulated by interrupts, calls, returns, and PUSH, PUSHF, POP, POPF instructions.

7. *ST, status register,* contains the status of the CPU.

8. *IE, interrupt enable register,* is used to enable or disable the DMA or CPU interrupts.

9. *IF, interrupt flag register*, is used to indicate whether a CPU interrupt is set or not.

10. *IOF, I/O flags register,* is used to control the I/O external pins.

11. *RC, repeat count register*, is used to specify how many times a block of code is to be repeated.

12. *RS, repeat start address register,* contains the starting address of the block of code to be repeated.

13. *RE, repeat end address register,* contains the ending address of the block of code to be repeated.

14. *PC, program counter,* contains the address of the next instruction to be fetched.

C. ADDRESSING MODES

Some of the speed of the C30 for DSP applications comes from the effective use of addressing. Different types of memory addressing are supported for program and data memory access. Some of the addressing types will be illustrated by examples using different instructions:

• *Register Addressing*: The operand is in a register. For example;

```
ADDI        R0, R1
```

adds the integer value in register R0 to that in register R1.

• *Immediate Addressing:* The operand is contained in the instruction. For example;

```
LDF         2.5, R1
```

loads the floating point number 2.5 into register R1.

• *Direct Addressing:* The operand is the address of a variable in memory, and the symbol "@" is used to indicate the direct addressing. In the following example,

```
SUBI        @808024h, R0
```

the integer value at the memory location 808024h is subtracted from the value in register R0.

• *Indirect Addressing:* The operand in indirect addressing is an auxiliary register containing the address of a variable in memory. Using the addresses stored in the registers AR0 - AR7, it is possible to access the memory. The symbol "*" is used to indicate the indirect addressing. In the following instruction,

```
LDI         *+AR0(3), R0
```

the number in the parenthesis is called the displacement. It can be any value between 0 and 255. Most of the time, IR0 and IR1 index registers are used as displacements. AR0 points to the start of the address, and +AR0(3) points to the fourth (AR0+3) address location from the beginning. The above instruction copies the integer value at the location pointed to by AR0+3 into register R0. The instruction,

```
SUBF        *-AR1(1), R1
```

subtracts the floating point value at the location pointed to by AR1-1 from the value in register R1. The instruction,

```
ADDF        *+AR1(IR0), R1
```

adds the floating point value at the location pointed to by AR1+IR0 to the value in register R1. IR0 is supposed to be loaded before this instruction. In the following example,

```
LDF         *AR0++, R0
```

the floating point value at the location pointed to by AR0 is loaded into the register R0, and AR0 is then incremented by one. After the instruction is executed , AR0 points to the location AR0+1, not AR0. In the instruction,

```
STF         R0, *--AR0
```

the location pointed to by AR0 is updated first by decrementing by one, and then the floating point value in register R0 is copied to the new location.

• *Bit-Reversed Addressing*: The bit-reversed addressing is used for reordering a scrambled sequence, especially in Fast Fourier Transform for proper sequencing of the data

to enhance the execution speed. The bit-reversed addressing mode is indicated by a "B" following the "++" symbol. The following program segment is an example of reordering a scrambled sequence (x0, x2, x1, x3) as (x0, x1, x2, x3):

```
        LDI     2, IR0
        RPTB    LOOP
        LDF     *AR0++(IR0)B, R0
LOOP    STF     R0, *AR1++
```

AR0 points to the scrambled array, and AR1 points to the reordered array. Index register IR0 is set to two, half of the array size. The block of code until the line specified by the LOOP word is repeated. At the first instruction, x0 is stored to the first location in AR1 via R0 register. After the execution, AR0 is updated in this way: AR0 points to the location (000), and is incremented by IR0 (010), (000 + 010) = (010). AR0 is updated to AR0+2 and points to the location which holds the x1. At the second instruction, x1 is stored to the second location in AR1 via R0 register. After the execution AR0 is updated; AR0 points to the location (010), and is incremented by IR0 (010). In the addition, the carry bit propagates to the right, not to the left. (010 +010) = (001), not (100). At the third instruction, AR0 points to the location of AR0+1, which holds x2. Then x2 is stored to the third location of AR1. AR0 is updated to (001+010) = (011). The next time AR0 points to the location of AR0+3, which holds x3, and is stored to the fourth location of the AR1. In order to make it clear, y bits are required to perform size N ($N= 2^y$) bit reversing. When y=2, the first and the second bit are swapped. If y=3, then the first and the third bit are swapped. For y=4, the first and the fourth and the second and the third bit would be swapped.

• *Circular Addressing:* It is used in some algorithms, such as convolution and correlation, to implement the delays. The symbol "%" is used to indicate the circular addressing, and it performs modular arithmetic. The size of the array is loaded into block size register BK. As new data is brought into one side of the array, the oldest data at the other end is discarded. The following code segment illustrates the use of circular addressing for the array {3, 5, 2, 6} pointed to by AR1.

```
LDI     4, BK
LDI     *AR1++(2)%, R0
LDI     *AR1++(3)%, R1
LDI     *AR1++(2)%, R2
LDI     *AR1++(1)%, R3
```

The array size four is loaded into the BK register. The first instruction loads 3 into register R0, and increments AR1 by two to point to 2 in the array. The second instruction loads 2 into register R1, and increments AR1 by three to point to 5. Because, 2 + 3 = 1 in modulo 4, the next instruction loads 5 into register R2, and increments AR1 by two to point to 6. The last instruction loads 6 into register R3, and increments AR1 by one to point to 3.

D. INSTRUCTION SET

There are 113 instructions in the C30 assembly language instruction set, and most of them are executed in a single cycle. These instructions are designed to support digital signal processing and numeric-intensive applications. Some of the instruction types are: Load and Store instructions, Two or Three Operand Math and Logical instructions, Input and Output instructions, Branch and Repeat instructions, and Parallel Operation instructions. Some simple examples are presented here to illustrate the use of these most common instructions.

• *Load and Store Instructions*: The 12 load and store instructions can load data into a register from another register or from a memory location, or store data from a register into the memory. Only registers R0-R7 are used for floating point instructions. For example,

```
LDF     R0, R1
```

loads the floating point value in register R0 into register R1,

```
LDI     1, AR0
```

loads the integer number 1 into register AR0,

```
LDF     *AR0, R0
```

loads the floating point value in the location pointed to by AR0 into register R0,

```
STF     R0, *AR0
```

stores the floating point value in register R0 to the location pointed to by register AR0,

```
STI          R0, @TOTAL
```

stores the integer value in register R0 into the location assigned to variable TOTAL.

• *Two or Three Operand Math and Logical Instructions:* The math and logical instructions are operated on two or three operands. If the source and destination operands are the same, it is not necessary to specify the operand two times. For two and three operand instructions, destination is always a register. The use of immediate or direct addressing is not allowed with three operand instructions. The instruction,

```
ADDI         *AR1++, AR0
```

adds the integer value in the location pointed to by AR1 to register AR0, and then AR1 is incremented by 1 to point to the next memory location AR1+1. However,

```
ADDI         3, R0, R1
```

is not a legal instruction, because immediate addressing is not allowed with three operand instructions. The three operand instruction,

```
MPYF3        R0, R1, R2
```

multiplies the floating point value in register R0 by the value in register R1, and stores the result in register R2. This instruction could also be written as

```
MPYF         R0, R1, R2
```

If the destination register is the same as the second source register, then it is not necessary to repeat the destination register again. It is also allowed to omit the number 3 after MPYF. The negation instruction,

```
NEGI         R0, R3
```

gets the negative value of the integer number in register R0 and stores the result in register R3. The absolute value instruction,

```
ABSI         R0
```

gets the absolute value of the integer number in register R0, and stores it again in register R0. The instruction,

```
SUBF         *AR0++, *AR1++, R2
```

subtracts the floating point value in the location pointed to by AR0 from the floating point value in the location pointed to by AR1, and stores the result in R2. Then, AR0 and AR1 are

incremented by one to point to the next memory location. The instruction

```
NOT     R1, R2
```

performs the bitwise logical complement of R1, and loads the result into R2. The instruction

```
OR      R0, R1
```

performs bitwise logical OR between R0 and R1, and loads the result into R1.

• *Input and Output Instructions:* An input data can be obtained, for example, from an oscilloscope, and after some mathematical operations on it, the result can be sent back to the oscilloscope for display. The following program block reads the data, multiplies it by two, and stores it:

```
LDI     @INADDR, AR0
LDI     @OUTADDR, AR1
FLOAT   *AR0, R0
LDF     2.0, R1
MPYF    R1, R0, R2
FIX     R2
STI     R2, *AR1
```

The first instruction loads the input address into AR0, and the second one loads the output address into AR1. The third instruction converts the integer value at the location pointed to by AR0 to its floating point equivalent and stores the result in R0. Then, the floating point number 2.0 is loaded into R1. The fifth instruction multiplies the floating point number in R0 by that in R1, and the result is stored in R2. The next instruction converts the floating number in R2 to its integer equivalent. The last instruction stores the integer value in R2 into the memory location pointed to by the output address.

• *Branch and Repeat Instructions:* The branch instruction, BR, causes the program to jump to a specified location. Branches can be conditional and are executed in four cycles. The decrement-and-branch instruction (DB), which is very useful in looping, decrements an AR register and jumps to the specified location until the AR register becomes negative. The following program segment executes the loop five times and loads the memory location pointed to by AR0 with {4, 8, 16, 32, 64}.

```
        LDI     4, AR0
        LDI     2, R0
LOOP    ADDI    R0, R0
        STI     R0, *AR1++
        DB      AR0, LOOP
```

The auxiliary register AR0 is loaded with 4 to execute the loop five times. Integer number 2 is loaded into R0. The loop starts, adds R0 to itself, and loads the result 4 to R0. Then, R0 is stored in the first location pointed to by AR1. AR0 is decremented by one and branched to the line LOOP. The second time in the loop, the integer number 4 in R0 is added to itself, and the result 8 is loaded into the second location pointed to by AR1. AR0 is decremented by two, and the loop is executed until the number in the AR0 register becomes negative.

Another type of branch instruction, delayed branch (BD), executes the next three instructions followed by the branch instruction before the branch instruction is executed. The result is four instructions in four cycles, or one-cycle delayed branch instruction instead of four-cycle branch instruction. For example, in the next program segment before the BD instruction is executed, LDI, ADDI and FLOAT instructions are executed respectively, and then the BD instruction takes place. In other words, before the program jumps to the line LOOP, the integer number in R1 is loaded into R0, integer 1 is added to the number in R0, and it is converted to its floating point equivalent; then the branch takes place.

```
        BD      LOOP
        LDI     R1, R0
        ADDI    1, R0
        FLOAT   R0
LOOP    .....
```

The Repeat Instruction is used for repeating the next instruction, or a block of code for a number of times. The two repeat instructions, RPTS (repeat single instruction) and RPTB (repeat a block of instructions), are executed in four cycles. The RPTS instruction must be loaded one less than the number of times the next instruction is to be repeated. In the following example, the first element of array {1, 2, 3, 4, 5}pointed to by AR0 is multiplied

by the second element, and the result is multiplied by the third one, and so on. The loop is executed four times, and the final result 120 is stored in R0.

```
        LDI     *AR0++, R0
        RPTS    3
        MPYI    *AR0++, R0
```

The RPTB instruction executes a block of code one more time than the value in the repeat counter register RC. The RPTB instruction loads the address of the first instruction in RS register and the address of the last instruction in RE register. The following example loads the integer number 1 to R1, and then loads the integer number 4 into RC to execute the loop five times. The first element of the array {1, 2, 3, 4, 5} pointed to by AR0 is multiplied by the integer number in R1, and the result is stored in R0. The number in R0 is stored in the memory location pointed to by AR1, and then the number in R1 is incremented by one. The loop is executed five times. After the last instruction, AR1 points to the memory location proceeded by {1, 4, 9, 16, 25} respectively, R0 holds 25, and R1 holds six.

```
        LDI     1, R1
        LDI     4, RC
        RPTB    LOOP
        MPYF    *AR0++, R1, R0
        STI     R0, AR1++
LOOP    ADDI    1, R1
```

• *Parallel Instructions*: The C30 is capable of executing multiply and add/subtract, load/store and arithmetic-store operations in parallel, thus increasing its performance to its peak value of 33.3 MIPS. The symbol "||" is used before the second operation to indicate the the parallel execution. There are some restrictions for parallel instructions, e. g., the multiply destination should be R0 or R1, and the addition destination should be R2 or R3; the updates of ARs should be zero, one, IR0 or IR1; both instructions should be the same type, integer or floating point. For example, the destination registers in the following example are R1 and R3 in the MPYF and ADDF instructions, respectively. The displacement of AR0 is one, and the displacement for AR1 is IR1.

```
    MPYF    R0, R4, R1
||  ADDF    *AR0++(1), *AR1++(IR1), R3
```

All registers are read at the beginning and loaded at the end of the execute cycle. If one of the parallel operations, ADDI in the following example, reads from a register, R0, and the operation being performed in parallel , MPYI, writes to the same register, then ADDI accepts as input the contents of R0 before it is modified by MPYI.

```
    MPYI    R1, *AR1, R0
||  ADDI    *AR0, R0, R2
```

In the next example, the integer number loaded from the address in AR0 and stored to the address in AR1 is not the same. The number in R0 at the beginning of the instruction execution is stored to the register AR1.

```
    LDI     *AR0++, R0
||  STI     R0, *AR1++
```

The following code updates the register AR0 first, then loads the contents into R1. At the same time loads the contents of AR1 into R2, and then updates AR1 by the number stored in IR0.

```
    LDF     *++AR0, R1
||  LDF     *AR1++(IR0), R2
```

Some arithmetic instructions can be executed in parallel with the store instruction. The following instruction negates the integer number pointed to by AR0, and at the same time stores the integer number in R0 (previous integer number) into the location pointed to by AR1. After the operations AR1 is incremented by one, and AR0 is incremented by the number stored in IR1.

```
    NEGI    *AR0++(IR1), R0
||  STI     R0, *AR1++
```

The next code subtracts the integer number in register R1 from that in register R3 and stores the result in R2 while storing the integer number in R0 into the location pointed to by AR2. After the operations, AR2 is decremented by one.

```
    SUBI    R1, R3, R2
||  STI     R0, *AR2--
```

The repeat instructions can be used with parallel instructions to increase the performance. The following code segment multiplies the elements of two arrays of size five.

```
    LDI     0, R1
    LDI     0, R3
    RPTS    4
    MPYI    *AR1++, *AR2++, R1
||  ADDI    R1, R3
    ADDI    R1, R3
```

Register R1 holds the result of multiplication, and R3 holds the result of addition. Both of the registers are loaded with zero at the beginning. The loop is executed five times. At the first parallel instruction, ADDI adds R1 to R3. The value in R1 is not the result of the first multiplication, but the value zero from initialization. So, the first result in R3 is zero. At the second multiplication, the result of the first multiplication is added to R3. At the end of the second multiplication, the result of the first multiplication is stored. That is why an extra addition instruction is placed after the parallel instruction. It adds the result of the last multiplication with the previous result.

E. PERIPHERALS

The C30 communicates with the outside world using on-chip peripherals. These on-chip peripherals include two timers, two serial ports, and an on-chip Direct Memory Access (DMA) controller. The peripherals are controlled by the peripheral registers located from 808000h to 809800h. The memory locations of these peripheral registers are shown in Figure 2-2 (a).

The two timers (Timer0 and Timer1) can be used to signal events at specified intervals to count external events or to generate interrupts. With an internal clock, the timer can be used to signal an external A/D converter to start its operation, or it can interrupt the DMA controller to begin a data transfer. With an external clock, the timer can count external events

and interrupt the CPU after a specified number of events. There are three registers associated with each timer. The memory locations of these registers are shown in Figure 2-2 (b). The Timer Global Control Register monitors the timer mode. The Timer Period Register specifies the signal frequency. The Timer Counter Register contains the current timer count. The counter resets to zero whenever it reaches the value in the period register.

The two serial ports, independent of each other, are used to transfer data in both directions. The clock for the ports can be generated either internally or externally. There are eight registers associated with each serial port.

The DMA Controller is used to perform I/O operations without interfering with the operation of the CPU. The DMA controller can read and write to any location in the C30 memory. There are four registers associated with the DMA controller.

The TMS320C30 digital signal processor has 16 million words of total memory space and an instruction cycle time of 60 nsec. It is a 32 bit processor and supports both fixed and floating point operations. The advanced architecture, rich instruction set, and the high speed make the TMS320C30 digital signal processor very suitable to implement the DSP applications in real time.

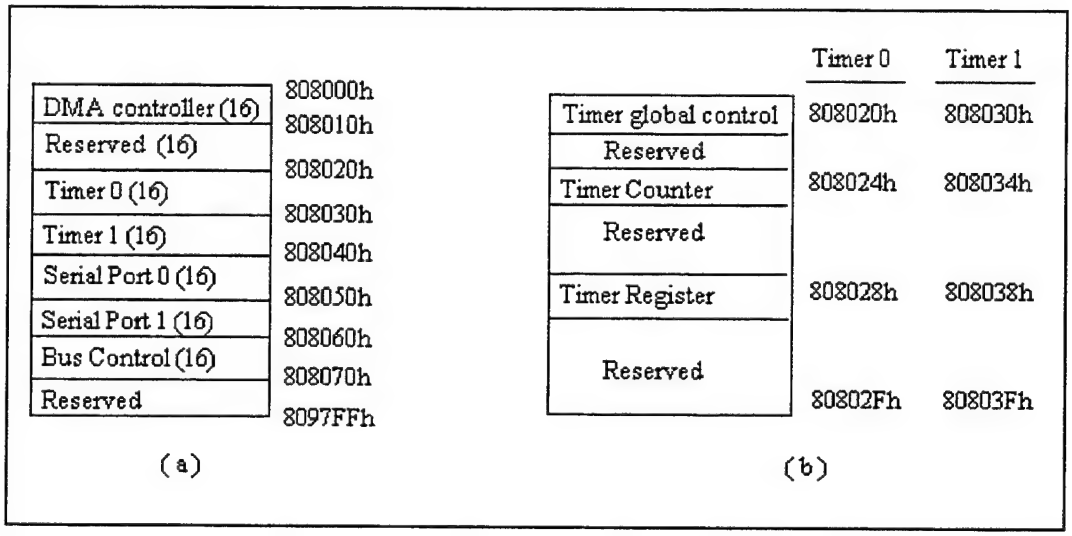

Figure 2-2 (a). Peripheral Memory Locations (b). Timer Memory Locations

III. THE TMS320C30 SYSTEM BOARD AND DEVELOPMENT TOOLS

The previous chapter provided background information about the architecture of the TMS320C30 processor. This chapter provides background information on the specific system at the Naval Postgraduate School. Chapter III is divided into two sections. The first section describes the C30 system board. The second section describes the development tools and the runtime support functions. More information about the C30 system board and the C compiler can be found in TMS320C30 System Board User's Guide [Ref. 6] and in TMS320 Optimizing C Compiler User's Guide [Ref. 3].

A. THE TMS320C30 BOARD

The TMS320C30 system board, designed by Loughborough Sound Images Ltd., comes with a package including PC-C30 board interface libraries, TI Assembler linker, TI C Compiler, connection cables (board to PC), and the necessary documentation. The C30 board used in this work is configured in microprocessor mode and has two A/D and two D/A converters, a parallel expansion system used as a memory mapped peripheral area, and two serial ports to transfer 8, 16, 24, or 32 bit data.

The C30 board is placed in a 16-bit slot in a PC-AT expansion bus. The maximum number of boards that can be used with a PC is limited by the number of slots available. All communication from PC to the C30 board is performed via the PC's I/O space. The address of the slot in the PC's I/O space where the board is connected is referred as the BASE address of the board, and by factory default it is 290h. If that address in the PC is connected to another device, it is necessary to change the base address on the board. The other recommended base addresses are 390h, 690h, and 790h, but they can be any address not used by the PC's I/O space. [Ref. 6]

Since the C30 is a 32 bit processor placed in a 16-bit slot and the floating number representation of the C30 board and the PC are different, interface routines are necessary to transfer the data between the C30 board and the PC. An interface library containing the functions to perform these conversions is avaliable. The following subsections describe the

communication or data transfer between the board and the PC and the interface libraries associated with it, the analog interfaces to obtain the data from other sources, and the interrupts to start/stop the input/output data transfer to/from the C30 memory.

1. Communication Between PC and C30 Board

Data is transferred to/from the C30 board via the 32-bit data registers in two steps because the board occupies a 16-bit slot in the PC I/O space. In order to write the data to the board, first, the least significant 16-bit word is written to the register at BASE+0. The most significant 16-bit word is then transferred to the register at address BASE+2. To read the data from the board, first, the least significant 16-bit word is read from the data register at address BASE+0. The upper 16 bit word is then read from the address BASE+2. [Ref. 6]

The PC can read from and write to any memory location in the C30 board, with the exceptions explained in Chapter II, but unless it is dual access memory (on the board), the C30 must be held during the PC accesses which results in extra execution time. So, it is better to use the dual access memory locations to move the data between the PC and the board. The C30 board is supplied with 64K of memory in this dual access area, locations from 30000h through 3FFFFh.

Another problem between the PC and the C30 board is the floating number representation. The C30 has its own floating number format, and the PC stores floating point numbers in the IEEE format. It is therefore necessary to convert the floating numbers to C30 or PC format before the transfer occurs.

The interface library makes these problems easier to deal with. The interface library allows us to write high level language programs to perform the DSP applications using the C30 board and the PC together. It includes the functions that downloads object modules to the C30 board, to start/stop the execution, and to move the data to/from the PC.

The source file for these interface functions is 30LIBRAR.C, and TMS30.H is the header file that includes the prototypes of the interface functions. The C source library is compiled with Borland C++ using the large memory model and put into library object format in LM30DEV.LIB. All the source files that use these interface functions should include the

TMS30.H header file and be linked with the LM30DEV.LIB library file. The names of the interface functions in the source file 30LIBRAR.C and a brief description for each function are listed in Table 3-1. [Ref. 6]

Table 3-1. Interface Functions

Function Name	Function Description
AssertReset	asserts the hardware reset signal to the C30 board.
AutoDec	puts the address counters of the C30 board into autodecrement mode.
AutoInc	puts the address counters of the C30 board into autoincrement mode.
CntrDis	disables the address counter of the C30 board interface.
CntrEnb	enables the address counter of the C30 board interface.
coffLoad	takes an object file and downloads to board memory.
GetFloat	converts floating point value from the C30 format to IEEE format.
GetInt	reads the integer value from the C30 memory location.
Get32Bit	takes the 32 bit value from the given C30 memory location.
Held	checks if the C30 board has held.
Hold	applies a hold request to the control register.
HoldAndWait	asserts hold to the chip and wait for acknowledgement.
loadArgs	is used to pass command line arguments to a program.
LoadObjectFile	downloads the object files from PC to the C30 memory
PutFloat	converts floating point value from IEEE format to the C30 format.
PutInt	puts an integer into the C30 memory location.
Put32Bit	puts the 32 bit value to a given location in the C30 memory.
RdBlkFlt	fills an array with flt. values from the given C30 memory locations.
RdBlkInt	fills an array with integer values from given C30 memory locations.
RdBlk32	reads the 32 bit values from the C30 memory into a given array.
ReadStatReg	reads the value of the Status Register.
RelReset	removes the hardware reset signal from the C30 board.
Reset	causes the C30 board to begin execution.
ResetCond	reads the state of the Reset bit from the Status Register.
SelectBoard	is used to initialize the board.
SetAddr	sets up the memory address register before a write or read.
SetCtrlReg	writes a new 8 bit value to the control register of the C30 board.
UnHold	removes the hold request from the control register.
UnHoldAndWait	releases a held and waits for acknowledgement.
WaitForHold	waits for the C30 to acknowledge a Hold request.
WaitForUnHold	waits for the C30 to acknowledge UnHolding of processor.
WarmSelect	is used to change from one C30 board to another in the PC.
WrBlkFlt	fills the C30 memory locations with flt.values from the given array.
WrBlkInt	fills the C30 memory locations with integer values from given array.
WrBlk32	fills the C30 memory locations with 32 bit values from given array.

The two C language programs, a C30 program 30XCOR.C and a PC program RXCOR.CP (see Appendix D), illustrate the utility of interface functions and transferring the data back and forth between the C30 board and the PC. This process is described in the following paragraph and in Table 3-2. The C30 board and the PC communicate by using "flags". PCPROCEEDFLAG and PCproceedflag point to the dual access memory location 30000h, and DSPPROCEEDFLAG and DSPproceedflag point to location 30001h. The PC program writes 1 to location 30001h if it wants the C30 program to start running, or the C30 program writes 1 to location 30000h if it wants the PC program to start running.

The PC program defines the board base address (0x290), two communication flags (PCPROCEEDFLAG and DSPPROCEEDFLAG), and two pointers for INPUT (30002h) and OUTPUT (30003h) arrays in the dual access memory area. The PC program starts and opens the data file DATA160.IN (holds the input data) and the result file COR.IN (to save the results), then reads the input data into "x" array. The PC program specifies the base address of the C30 board (0x290) and downloads the C30 program by "loadstat", and then sets the PCPROCEEDFLAG and DSPPROCEEDFLAG to 0 . The C30 program starts with "Reset", and assigns pointers PCproceedflag, DSPproceedflag, and x array to the Comm0 (30000h), Comm1(30001h), and Comm2 (30002h) memory locations (defined in LSICMAP.CMD file), respectively. At the same time PC program waits for PCPROCEEDFLAG to be set to 1 (PCPROCEEDFLAG-PCproceedflag and DSPPROCEEDFLAG-DSPproceedflag pairs point to the same memory location, so PC/DSPflag is going to be used for both cases). The PCflag is set to 1, and the C30 program waits until the DSPflag is set to 1. The PC program gets the starting address by "inloc = Get32Bit()" and downloads the input "x" array into the memory location starting at 30002h (specified by INPUT) by "WrBlkFlt", then sets the DSPflag to 1 and waits for the PCflag to be set to 1. The C30 program initializes the Timer 1 registers, and the counter register starts counting. A call to function "xcor" is performed, and then Comm3 takes the starting address of the result array. PCflag is set to 1, and the PC program gets the starting address of the result by "outloc = Get32Bit()" and downloads the output "ac" array into the memory starting from location 30003h (specified by OUTPUT) by "RdBlkFlt", and then displays the results on the PC screen and saves the results in COR.IN file.

Table 3-2. Program Flow Between PC and C30 Board.

PC PROGRAM	DSP PROGRAM
Starts running, open files, selects the board address, downloads the DSP program, resets the DSP program and waits for the PCflag.	Waits to be reset.
Waits for the PCflag.	DSP program starts running, assigns pointers to PC/DSPflags and input array. Sets the PCflag=1, and waits for the DSPflag.
Downloads the input array into memory, sets the DSPflag=1, and waits for the PCflag.	Waits for the DSPflag.
Waits for the PCflag.	Makes the computation, puts the result into the memory, sets the PCflag=1, and waits for the DSPflag (if necessary).
Gets the output array, prints them on the screen, and saves them in a file. Sets the DSPflag=1 (if necessary).	Waits for the DSPflag.

Note that, while PC or C30 is running, the other one is waiting for its turn for some amount of time. This is done in order to transfer the input array to the C30, or to transfer the output array to the PC in time. Otherwise, the C30 would start computing before it received the input array, and would cause the function to use the leftover values in the memory which would result in wrong output values. It is not always necessary to wait if there is something the processor can do that is independent of the PC.

2. Analog Interfaces

External devices (oscilloscope, microphone, etc.) can be connected directly to the ports on the C30 board, and using the A/D or D/A converters (ADC or DAC) on the board, signals from the external devices can be sampled, processed and the result sent back to the devices or to a PC for further applications. In order to use the analog signal I/O capabilities

of the board, output of the signal source must be in the range of ±3V. Signals from microphones require amplification. The outputs will not directly drive a loudspeaker, but can be monitored on an oscilloscope.

There are two input and two output channels to interface with the analog signals. Three registers connected to the C30 expansion bus are used to access the ADC/DACs on the two analog I/O channels. Their address locations are:

1. Read / write Channel A A/D and D/A: 804000h.
2. Read / write Channel B A/D and D/A: 804001h.
3. Generate Software Conversion Trigger: 804008h.

To input data from ADC on channel A, the value at the location 804000h is read, and to output the data to DAC on the same channel, the data is moved to the same location. The location 804008h accesses a register which can be used externally to start an A/D or D/A conversion. [Ref. 1, 6]

Conversions on both channels are initiated by either an external trigger signal or an on-chip timer. The following example describes how to set up on-chip peripheral Timer 1:

In order to set up Timer 1 in the correct mode, 6C1h is written to the location 808030h, which enables Timer 1 to use internal clock. The sample rate is set by writing a value to the Timer 1 period register at location 808038h. The following formula is used to find the value to be loaded to the Timer 1 period register [Ref. 6]:

The value to be loaded = Period in microseconds / 0.12

where the period is the desired sampling period time (in microseconds) between samples. The value in the Timer 1 counter register is continuously incremented starting from zero once every 120 ns. When the value in the counter register becomes equal to the value in the period register, a pulse initiates the A/D or D/A conversion and loads a zero starting count value to the counter register. For example, to have a sampling rate of 8192 Hz, 3F9h or 1017 is loaded into the period register.

3. Interrupts

Interrupts are used to allow the processor to respond to the external as well as the internal events. The first twelve locations of the memory are reserved for the hardware interrupt vectors. These locations contain the address of an interrupt service routine which is to process the corresponding interrupt when it occurs. For the C30 to respond to interrupts, the appropriate bits in the ST and IE registers are set to 1. When the corresponding bit in the IE register is set and interrupts are enabled by setting the global interrupt register (GIE) bit in the ST register to 1, interrupt process begins.

Interrupt service routines can be written in C language, and interrupts are enabled by using inline assembly language statements. C interrupt functions have names with this format: "c_intnn ( )" where nn is a two digit number between 00 and 99. A C interrupt routine like any other C function can have local variables and register variables. Interrupt names, corresponding sections, memory locations and their functions are described in Table 3-3. [Ref. 1,6]

The C programs in Appendix E illustrate the use of ADC/DACs, and interrupt process to input and output data transfer. The input is a 1kHz sine wave (it can be any type of signal). There are two outputs: The first one is the sampled version of the input signal, and the second one is the same as the first one with a different amplitude.

The sampling rate 8192 Hz is set by the Timer 1 registers as explained in the previous subsection. The interrupt is enabled by the following two inline assembly language statements.

```
asm ("    OR  2h, IE ");
asm ("    OR  2000h, ST ");
```

The first statement sets the second bit in the IE register, corresponding to INT1 which enables the interrupt from Timer 1.The second statement sets the global interrupt enable bit (bit 13) in the C30's status register. This bit must be set, or the C30 will not respond to any interrupts even if the IE register is enabled.

There are three inline assembly language statements after the main program:

```
asm ("  .sect   \".int02\"");
asm ("  .word   _c_int02");
```

asm (" .text ");

The first statement tells the assembler to locate the program .int02 in the vector table. The memory locations of the vector table and their corresponding section names are defined in the map file LSICMAP.CMD, so it is essential to link the map file. The second statement contains the name of the interrupt service routine and tells the assembler to insert the vector to interrupt service routine. The third statement tells the assembler to put the rest of the code in the text code section. [Ref. 6]

Table 3-3. Interrupt Names and Memory Locations

Interrupt Name	Section Name	Location	Function
Reset	.int00	0h	External interrupt (Power on)
INT0	.int01	1h	External interrupt 0 (ADCs)
INT1	.int02	2h	External interrupt 1
INT2	.int03	3h	External interrupt 2
INT3	.int04	4h	External interrupt 3
XINT0	.int05	5h	Internal serial port 0 transmit interrupt
RINT0	.int06	6h	Internal serial port 0 receive interrupt
XINT1	.int07	7h	Internal serial port 1 transmit interrupt
RINT1	.int08	8h	Internal serial port 1 receive interrupt
TINT0	.int09	9h	Internal timer 0 interrupt
TINT1	.int10	Ah	Internal timer 1 interrupt
DINT	.int11	Bh	Internal DMA interrupt

The value in the Timer 1 counter register is incremented once every 120 nsec, and when the value in the counter register is equal to or bigger than the value in the period register (3F9h or 1017), it resets itself to zero and initiates a pulse for A/D conversion. The A/D channel will perform the conversion and output an end-of-convert signal, which is linked

to the C30 as the INT1 interrupt request. Then the interrupt service routine, c_int02 ( ), will read the input from channel A, multiply the input by a constant and write the result in channel B, and write the input to channel A as output. Interrupts occur every 122 microsec.

B. DEVELOPMENT TOOLS

Whether the programs are written in C language or in assembly language, it is necessary to link them with the interface and runtime support libraries to obtain the executable files. This section describes the development tools (including compiler, assembler, linker and library built utilities) and how to invoke them with the required options and runtime support libraries. Although it is possible to invoke the compiler, assembler and linker individually, the cl30 compiler shell is introduced to perform all three of them in a single step.

1. Compiler

The TMS320 floating point C compiler produces the assembly codes from the C source code and conforms to the ANSI C standard. The compiler is made up of three separate programs: parser, optimizer and code generator. These programs can be invoked individually, but cl30 shell program invokes all three of them automatically. The cl30 program shell compiles, assembles and links the source files in a single step to obtain the executable file. Figure 3-1 shows the path cl30 takes. The general format for invoking the cl30 shell is [Ref. 3]:

cl30 [-options] [input filenames] -z [-link options]

The input file names can be C files, assembly files, or object files. Files without extensions are assumed to be C files. If the -z is used, cl30 shell invokes the linker; however, it is optional. The options control the operation of the compiler. Some of the options are listed below:

-c: compile and assemble, disable linking (negates -z).

-z: enable linking.

-n: compile only.

-s: interlist C and asm statements; it inserts C source code as comments.

-o: optimizes the code for efficiency; it is described in the next chapter.

-mb: enables the big memory model.

-al: produces an assembly listing file.

Linker options will be explained in the linker subsection.

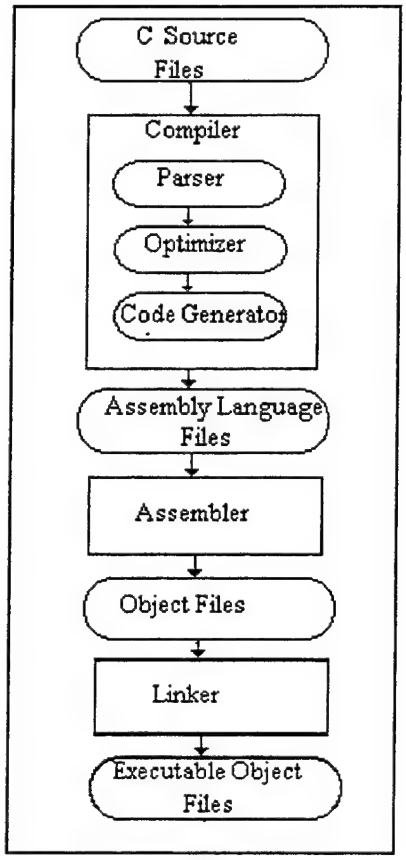

Figure 3-1. The Path of cl30 Command

2. Assembler

The assembler translates the assembly language files into machine language files. The general format for invoking the assembler is [Ref. 3]:

asm30 [input file [object file [listing file]]] [-options]

The input file names are the assembly language file names. The generated object and listing file names get the same name as the source file with the extensions .obj and .lst, respectively, unless specified otherwise. Some of the options are listed below:

-l: produces a listing file.

-mb: enables the big memory model.

3. Linker

By using the cl30 shell, we can compile, assemble and link in one step. It is also possible to link in a separate step, especially if we have multiple source files. We can compile and assemble individual files, and then link them together. The general format for invoking the linker is [Ref. 3]:

lnk30 [-options] object files

The object files are the files to be linked; some of the options are:

-c: links the C code; the static data is copied from ROM into RAM.

-cr: links C code; the static data is loaded directly into RAM.

-heap xxxx: sets heap size to xxxx words, e. g., -heap 4096; the default is 1K.

-stack xxxx: setsstack size to xxxx words, e. g., -stack 4096; the default is 1K.

-i dir: defines the library search path if the libraries defined in -l option are not in the current directory.

-l lib: links library files, e. g., -l rts30b.lib links the library rts30b.lib with the object files.

-o: generates an executable output file, e. g., -o rxcor names the output file as rxcor.out; if no names are specified, the default is a.out.

4. Runtime Support Functions and Library Built Utilities

The TMS320C30 compiler includes a library that contains the ANSI standard runtime support functions. This library contains the standard runtime support functions, compiler utility functions and math functions that can be called from C programs that have been compiled for the C30. All the functions are declared in the source file *rts.src*. This file contains all runtime support functions and the mathematical functions. Some of the mathematical functions (such as division and 24 bit integer multiplication) in the source file rts.src have been written in assembly language to speed up the execution.

The header files that declare the runtime support functions (assert.h, ctype.h, errno.h, float.h, limits.h, math.h, stdarg.h, setjmp.h, stdlib.h, string.h, and time.h) must be included in

the main program if the program calls one or several of the functions declared by the header file. The functions and the header files are described in detail in TMS320 Optimizing C Compiler User's Guide, [Ref.3].

If the program uses one of these functions that are described in rts.src, object file of the program should be linked with the library created from the rts.src source file. The TMS320 C compiler contains a library file, *rts30.lib*, built from the source file rts.src with the highest optimization level (-o). This library can be linked with any file compiled for the C30 with the small memory model. The compiler supports two memory models: the small memory model enables the compiler to access the memory by restricting the global data space to a single 64K word data page; the big memory model allows unlimited space, and the data can be placed anywhere in the memory. [Ref. 3]

If the code is compiled with the large memory model, then it is essential to create another library from the source file. The library built utility is used to custom build runtime support libraries. The general format to invoke the library built utility mk30 is [Ref. 3]:

mk30 [- options] [source library name] -l [object library name]

For instance, the following example builds the standard runtime support library rts.src as a library named rts30b.lib. The library is compiled for the C30 (-v30) with highest optimization (-o2 or -o) according to the big memory model (-mb). The example assumes that the runtime support header files are in the current directory (--u).

```
mk30  --u -v30 -o -mb rts.src -l rts30b.lib
```

We can also create our own object libraries. The C30 archiver allows us to collect individual object files into a single file called an archiver or a library. Once an archiver has been created, new files can be added, deleted, or modified. This archiver (library) can be used as a linker input during the compilation. The general format to invoke the archiver is [Ref 3]:

ar30 [-option] library name [file names]

Only one option can be used with each invokation. The library name is the desired name for the newly created library. The file names are the object files to be combined. Some of the options are:

-a: adds the specified file to the library; if there is another file with the same name, this option does not replace it, thus we may have several files with the same name.

-d: deletes the files from the library.

-r: replaces the files in the library; if the specified file is a new one, the archiver adds it in the library.

The following example creates a library called dsplib.lib which combines the files xcor.obj, burg.obj, and schur.obj:

```
ar30  -a dsplib.lib xcor.obj burg.obj schur.obj
```

If it is necessary to make some changes in one of the subroutines and rebuild the library, first of all, after the modification is done, the function is compiled again to obtain the object file. Then, -r option is used to rebuild the library. The next example replaces or modifies the file xcor.obj with the old one and adds the rctopc.obj file to the library:

```
ar30  -r dsplib.lib xcor.obj rctopc.obj
```

The C30 has a powerful C compiler and development tools to compile and link the C language or assembly language programs. The compilation process is easy and performs multiple tasks in a single step.The interface library allows the programmer not to think about the details of the data transfer between the PC and the C30 board. Analog interfaces on the chip and interrupts enhance the capability of the C30 board to input/output the data from/to outside sources.

IV. AUTOREGRESSIVE PARAMETER ESTIMATION

A. LINEAR PREDICTION

Linear prediction is based on estimating the current sample of a given input signal from the linear combinations of its past samples. Consider a zero-mean, stationary, real signal x[n]. The estimate or the prediction of this signal takes the form

$$\hat{x}\ [n] \ = \ - \ \sum_{k=1}^{M} \ a_k \ x[n-k] \ . \qquad (1)$$

This is referred as the M-th order predictor, and the coefficients $\{ \ a_1, a_2, \ldots a_M \ \}$ are called the *predictor coefficients*. We define the prediction error, e[n], as

$$e[n] \ = \ x[n] \ - \ \hat{x}[n] \ . \qquad (2)$$

We want to find the optimal predictor coefficients. The prediction coefficients are chosen to minimize the power of the prediction error,

$$\sigma_e^2 \ = \ \mathscr{E} \ [\ | \ e[n] \ |^2 \ ] \ , \qquad (3)$$

where $\mathscr{E}$ is the expectation operator, and σ_e^2 is the *prediction error variance*. Applying the orthogonality principle which states that the error is orthogonal to the previous observations, we can write [Ref. 7]

$$\mathscr{E}\,[\,x[n-k]\;e[n]\,] = \mathscr{E}\,[\,x[n-k]\;(\,x[n] - \hat{x}[n]\,)\,] = 0\;,\quad k=1, 2, ..., M, \qquad (4)$$

and this leads us to

$$R_x[k] = -\sum_{j=1}^{M} a_j\, R_x[k-j]\;, \qquad k= 1, 2, ..., M, \qquad (5)$$

where $R_x[k]$ are the autocorrelation lags of the signal x[n]. The minimum prediction error variance can be obtained from

$$\sigma_e^2 = \mathscr{E}\,[\,x[n]\;e[n]\,] = R_x[0] + \sum_{k=1}^{M} a_k\, R_x[-k] \qquad (6)$$

Using the symmetry property of the Toeplitz autocorrelation matrix ($R_x[k] = R_x[-k]$) and combining Equations (5) and (6) into (M+1) × (M+1) matrix equations, we obtain

$$\begin{bmatrix} R_x[0] & R_x[1] & \cdots & R_x[M] \\ R_x[1] & R_x[0] & \cdots & R_x[M-1] \\ \vdots & \vdots & \vdots & \vdots \\ R_x[M] & R_x[M-1] & \cdots & R_x[0] \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ \vdots \\ a_M \end{bmatrix} = \begin{bmatrix} \sigma_e^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \qquad (7)$$

These equations are known as the *normal equations* of linear prediction. They provide the solution to both signal modeling and linear prediction problems. They help determine the prediction parameters { $a_1, a_2, ..., a_M\,;\, \sigma_e^2$ } of the signal x[n] directly in terms of the correlation function $R_x[k]$. [Ref. 7]

The prediction error filter in Figure 4-1 is an FIR filter with a transfer function of

$$A(z) = 1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_M z^{-M} \tag{8}$$

which produces e[n] from the input x[n]. As M increases, more past information is taken into account, and we expect the prediction of x[n] to become better, thus yielding a smaller prediction error.

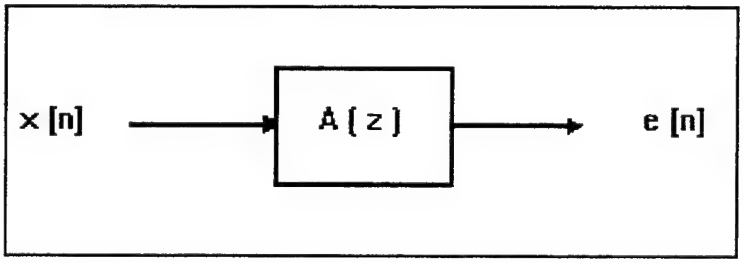

Figure 4-1. Linear Prediction Error Filter

Equation (1) predicts the current value of a signal based on the M previous samples. We can also predict the earlier sample of a signal using M samples in the future of the sample being predicted. Then the estimate takes the form

$$\hat{x}[n-M] = - b_1 x[n-M+1] - b_2 x[n-M+2] - \ldots - b_M x[n] \ , \tag{9}$$

where b_i are the *backward prediction coefficients*, and if we follow the same procedure as we did above, we conclude that for real random processes, forward and backward AR models are statistically identical. The prediction parameters a_i and b_i are the same as well as the prediction error variance σ_e^2. [Ref. 7]

B. AUTOREGRESSIVE MODELING

Autoregressive (AR) modeling is a very powerful approach for signal modeling. This is because accurate estimates of the AR parameters can be found by solving a set of linear equations. Suppose it is desired to represent the signal x[n] by an AR (M) model. In AR modeling, we attempt to predict x[n] on the basis of the previous M successive values of w[n], a zero-mean, white noise with variance $\sigma_w^2 = \sigma_e^2$. This excitation noise signal produces the desired signal x[n] through an IIR filter in Figure 4-2 with a transfer function of 1 / A(z).

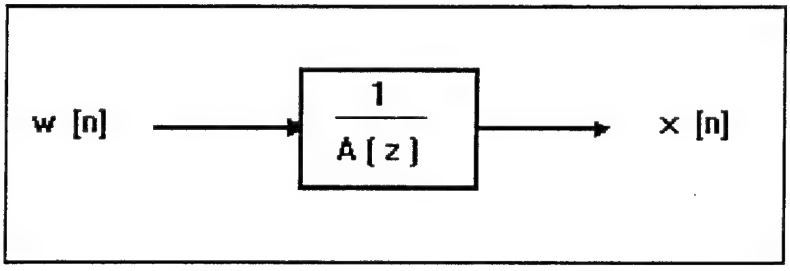

Figure 4-2. AR Model of Prediction Error Filter

We can write the system output x[n] as the linear combination of its past samples and the excitation noise as follows:

$$x[n] = - a_1 x[n-1] - a_2 x[n-2] - \ldots - a_M x[n-M] + w[n] \ , \qquad (10)$$

and proceeding the same way we did in finding the normal equations in Equations (2) - (6), we obtain the following relation for AR models

$$\begin{bmatrix} R_x[0] & R_x[1] & \cdots & R_x[M] \\ R_x[1] & R_x[0] & \cdots & R_x[M-1] \\ \vdots & \vdots & \vdots & \vdots \\ R_x[M] & R_x[M-1] & \cdots & R_x[0] \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ \vdots \\ a_M \end{bmatrix} = \begin{bmatrix} \sigma_w^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix} . \qquad (11)$$

These equations are known as the *Yule-Walker equations*. We see that the above autocorrelation matrix, $R_x[k]$, is Toeplitz since the elements along any diagonal are identical. Also the matrix is positive definite which follows from the positive definite property of the autocorrelation function [Ref. 7]. In order to find the AR parameters $\{ a_1, a_2, ..., a_M, \sigma_e^2 \}$, one must solve Eq. (11) with the M+1 autocorrelation lags $R_x[0], R_x[1], ..., R_x[M]$. These equations are identical to Eq. (7). The optimal linear prediction coefficients are the AR parameters, and the minimum prediction error variance is just the excitation noise variance. This will only be true if the order of the AR process and the order of the linear predictor are the same. Therefore, AR parameter identification and linear prediction of an AR (M) process yield identical results. We notice that by choosing a sufficiently large order M, we can always obtain the exact parameter values when x[n] is an AR (M) process.

In the normal or Yule-Walker equations, we simply replace the autocorrelation $R_x[k]$ by the corresponding autocorrelation lags computed from the given block of input data,

$$\hat{R}_x[k] = \frac{1}{N} \sum_{n=0}^{N-1-k} x[n]\, x[n+k] , \qquad \text{for } k = 0, 1, ..., M , \qquad (12)$$

where only the first M+1 lags are needed in the autocorrelation matrix with the condition of $M \leq N - 1$.

In practice, the normal equations provide a means of determining approximate estimates for the model parameters { $a_1, a_2, ..., a_M$; σ_e^2 }. Typically, a block of length N of recorded data is available { x[0], x[1], ..., x[N-1] }. There are many different methods of extracting reasonable estimates of the model parameters using this block of data.

C. LEVINSON ALGORITHM

The Levinson algorithm (sometimes referred as Levinson-Durbin algorithm) is a computationally efficient way to find the prediction and reflection coefficients and prediction error variance from the autocorrelation function. It can also be used to find the appropriate model order to reduce σ_e^2 to a desired value when the correct model order is not known.

In order to find the AR parameters, the Yule-Walker or Normal equations must be solved. These equations can be solved by the Gaussian elimination method which requires $O(M^3)$ operations or by the Levinson algorithm which requires $O(M^2)$ operations.

The solution of Yule-Walker equations,

$$\begin{bmatrix} R_x[0] & R_x[1] & \cdots & R_x[M] \\ R_x[1] & R_x[0] & \cdots & R_x[M-1] \\ \vdots & \vdots & \vdots & \vdots \\ R_x[M] & R_x[M-1] & \cdots & R_x[0] \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ \vdots \\ a_M \end{bmatrix} = \begin{bmatrix} \sigma_e^2 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad (13)$$

provides the optimal set of coefficients to predict the x[n] as a linear combination of M past samples. If we wish to find not only the M-th order linear predictor coefficients, but also the predictor coefficients of the orders { M-1, M-2, ..., 1 }, we need to solve Eq. (13) for each order separately. Since this procedure is computationally burdensome, we need to find an approach to compute the M-th order parameters recursively using the (M-1)-st order parameters.

The Levinson algorithm is based on estimating the AR parameters of order p from the parameters of order p-1. The Levinson algorithm recursively computes the parameter sets { a[1,1]; σ_1^2 }, { a[2,1], a[2,2]; σ_2^2 }, ..., { a[M,1], a[M,2],..., a[M,M]; σ_M^2 } where a[p,k] is the k-th coefficient of the p-th order predictor. It is important to note that { a[p,1], a[p,2], ..., a[p,p] ; σ_p^2 } as obtained from the Levinson algorithm is the same as that would be obtained by solving the Yule-Walker equations with M = p.

The algorithm [Ref. 7, 10] is as follows:

STEP 0. $\mathbf{a[0] = 1}$; $\quad a[1,1] = -\dfrac{R_{xx}[1]}{R_{xx}[0]}$; $\quad \sigma_1^2 = (1-|a[1,1]|^2)\, R_{xx}[0]$

STEP 1. $\quad \mathbf{p = 2, \ldots, M}$

STEP 2. $$a[p,p] = -\frac{R_{xx}[p] + \sum_{n=1}^{p-1} a[p-1,n]\, R_{xx}[p-n]}{\sigma_{p-1}^2} \qquad (14)$$

STEP 3. $$a[p,i] = a[p-1,i] + a[p,p]\, a[p-1,p-i] \qquad i = 1, 2, \ldots p-1 \qquad (15)$$

STEP 4. $$\sigma_p^2 = (1 - |a[p,p]|^2)\, \sigma_{p-1}^2 \qquad (16)$$

STEP 5. **Go to step 1 and increase the order by 1**

In the algorithm, step 0 is the initialization, and steps 1-5 are the recursion steps. Steps 1-5 are computed for p = 2 to M where a[p,p]'s are called the *reflection coefficients* and denoted by k[p], and σ_M^2 is the noise variance. The Levinson algorithm provides the AR parameters for all order (1, 2, ..., M) AR models which is a useful outcome when we do not know a priori the correct model order.

Some of the properties of the Levinson algorithm assuming that the process is an AR(M) process are: [Ref. 7]

1. a[M+1,p] = a[M,p] for p = 1, 2, .., M and a[M+1,M+1] = k[M+1] = 0.

2. a[p,p] = k[p] = 0 for p > M and hence $\sigma_p^2 = \sigma_M^2$ for p > M.

3. The property that | k[p] | < 1 leads to $\sigma_{p+1}^2 \leq \sigma_p^2$, which implies that σ_p^2 first gets its minimum value at the correct model order.

4. If the process consists solely of p sinusoids, the recursion must terminate when | k[p] | =1; because at step 4, σ_p^2 will be equal to 0. Since, in practice, signals are corrupted with noise, no attention was paid to this condition during the implementation of the algorithm.

Given the reflection coefficients and forward and backward prediction errors for order p, we can obtain the forward and backward prediction errors for the order p+1 without solving the normal equations.

The lattice filter representation, which can be used to compute the forward and backward prediction errors for the p-th order predictor based on the (p-1)-st order predictor and the reflection coefficients, k[p], is as follows: [Ref. 7]

$$e^f[p,n] = e^f[p-1,n] + k[p]\, e^b[p-1,n-1] , \qquad (17)$$

$$e^b[p,n] = e^b[p-1,n-1] + k[p]\, e^f[p-1,n] , \qquad (18)$$

where $e^f[0,n] = e^b[0,n] = x[n]$ since no prediction is made, and $e^f[p,n]$ is the p-th order forward and $e^b[p,n]$ is the p-th order backward prediction errors. The lattice filter is shown in Figure 4-3 and is equivalent to the standard prediction error filter.

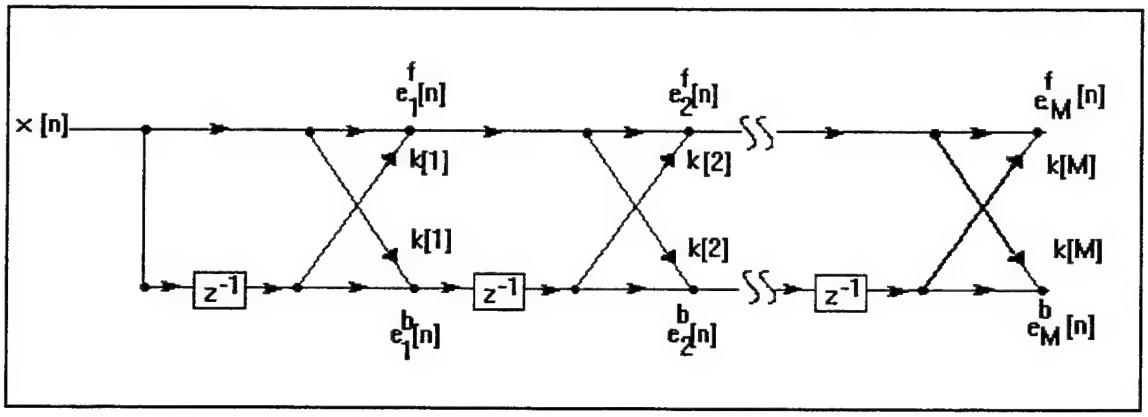

Figure 4-3. Lattice Realization of Prediction Error Filter

D. BURG ALGORITHM

One of the most popular approach to AR parameter estimation was introduced by Burg. Burg minimized the sum of the forward and backward prediction powers as opposed to the Levinson algorithm where only the forward prediction power was minimized. Burg algorithm guaranties a stable all-pole filter, and the magnitudes of the reflection coefficients are less than one. [Ref. 8]

The Burg method estimates reflection coefficients directly from the data samples without estimating the autocorrelation values, and then uses the Levinson algorithm to estimate the AR parameters. In the Burg method, reflection coefficients for order p ,k[p], are estimated by minimizing the sum of the forward and backward prediction error powers,

$$\rho[p] = \sum_{n=p}^{N-1} \left(| e^f[p,n] |^2 + | e^b[p,n] |^2 \right) . \qquad (19)$$

Substituting Equations (17) and (18) into Eq. (19), differentiating with respect to k[p], setting the derivative to zero, and solving for the reflection coefficients, we obtain,[Ref. 7]:

$$k[p] = \frac{-2 \sum_{n=p}^{N-1} e^{f}[p-1,n]\ e^{b}[p-1,n-1]}{\sum_{n=p}^{N-1} \left(|\ e^{f}[p-1,n]\ |^2 + |\ e^{b}[p-1,n-1]\ |^2 \right)} \quad . \tag{20}$$

In order to reduce the computations in the denominator of Eq. (20), we can use the following recursive relationship:

$$DEN[p] = DEN[p-1]\ (\ 1 - |k[p]|^2\ ) - |\ e^{f}[p-1,p-1]\ |^2 - |\ e^{b}[p-1,N-1]\ |^2 \quad . \tag{21}$$

Burg algorithm [Ref. 7, 10] can be summarized as follows:

STEP 0. $\quad R_x[0] = \frac{1}{N} \sum_{n=0}^{N-1} |\ x[n]\ |^2 \quad ; \quad \sigma_0^2 = R_x[0]$

$$e^{f}[0,n] = x[n] \qquad n = 1, 2, ..., N-1$$

$$e^{b}[0,n] = x[n] \qquad n = 0, 1, ..., N-2$$

STEP 1. $\quad p = 1, ..., M$

STEP 2. $$k[p] = \frac{-2 \sum_{n=p}^{N-1} e^{f}[p-1,n]\ e^{b}[p-1,n-1]}{\sum_{n=p}^{N-1} \left(|\ e^{f}[p-1,n]\ |^2 + |\ e^{b}[p-1,n-1]\ |^2 \right)} \tag{22}$$

STEP 3. $$a[p,i] = a[p-1,i] + k[p]\ a[p-1,p-i] \qquad i = 1, 2, \ldots p-1 \qquad (23a)$$

$$a[p,i] = k[p] \qquad i = p \qquad (23b)$$

STEP 4. $$\sigma_p^2 = (1 - | k[p] |^2)\ \sigma_{p-1}^2 \qquad (24)$$

STEP 5. $$e^f[p,n] = e^f[p-1n] + k[p]\ e^b[p-1,n-1] \qquad n = p+1, p+2, \ldots, N-1 \qquad (25a)$$

$$e^b[p,n] = e^b[p-1,n-1] + k[p]\ e^f[p-1,n] \qquad n = p, p+1, \ldots, N-2 \qquad (25b)$$

STEP 6. **Go to step 1 and increase the order by 1**

In the algorithm, step 0 is the initialization, and steps 1-6 are the recursion steps. Steps 1-6 are computed for p = 1 to M. The reflection coefficients are computed in step 2, prediction coefficients in step 3, noise variance in step 4, and updates of forward and backward prediction error powers in step 5. Computations in step 5 are not necessary at the last order.

E. SCHUR ALGORITHM

In step 2 of the Levinson algorithm, one has to compute the inner product which is hard to implement on a parallel pipelined computer architecture. Schur algorithm [Ref. 10] is an efficient alternative method to Levinson algorithm and can be used to compute the reflection coefficients from the given autocorrelation lags without computing any inner products.

We define the forward and backward gapped [Ref. 9] functions which are the crosscorrelation functions between the prediction error and the signal, of order p,

$$g^f[p,k] = \mathcal{E}\ [\ e^f[p,n]\ x[n-k]\ ] = R_{e^f[p]\ x}[k]\ , \qquad k = 0, 1, \ldots, p, \qquad (26)$$

and substituting Eq. (2) into Eq. (26), we obtain the following for the forward gapped function

$$g^{f}[p,k] = \sum_{l=0}^{p} R_x[l-k]\ a[p,k] \ , \qquad k = 0, 1, ..., p, \qquad (27)$$

and following the same procedure for backward gapped function, we obtain

$$g^{b}[p,k] = \mathcal{E}\ [\ e^{b}[p,n]\ x[n-k]\ ] = R_{e^{b}[p]\ x}[k] \qquad (28)$$

or

$$g^{b}[p,k] = \sum_{l=0}^{p} R_x[p-l-k]\ b[p,k] \ , \qquad k = 0, 1, ..., p, \qquad (29)$$

where b[p,k] = a[p,p-k], and the backward gapped function is the reflected and delayed version of the forward one,

$$g^{b}[p,k] = g^{f}[p,p-k] \ . \qquad (30)$$

Therefore, we can now say that

$$g^{f}[p,k] = 0 \qquad k = 1, 2, ..., p \qquad ; \qquad g^{b}[p,k] = 0 \qquad k = 0, 1, ..., p-1. \qquad (31)$$

Inserting Eq. (17) into Eq. (26) and Eq. (18) into Eq. (28), we obtain the lattice recursion of gapped functions for Schur algorithm,

$$g^{f}[p+1,k] = g^{f}[p,k] - k[p+1]\ g^{b}[p,k-1] \ , \qquad (32)$$

$$g^{b}[p+1,k] = g^{b}[p,k-1] - k[p+1]\ g^{f}[p,k] \ . \qquad (33)$$

Eq. (4) can be rewritten in the form of crosscorrelation between the signal and its past forward and backward prediction errors,

$$\mathcal{E}\,[\,x[n-k]\;e[n]\,] = \mathcal{E}\,[\,(e^{f}[p-1,n]-a[p]\;e^{b}[p-1,n-1])\;x[n-k]\,] = 0\;, \qquad (34)$$

or

$$R_{e^{f}[p-1]\,x}[k] - k[p]\;R_{e^{b}[p-1]\,x}[k-1] = 0 \qquad k = 1, 2, ..., p. \qquad (35)$$

Eq. (35) can be solved for the reflection coefficients

$$k[p] = \frac{R_{e^{f}[p-1]\,x}[p]}{R_{e^{b}[p-1]\,x}[p-1]} = \frac{g^{f}[p-1,p]}{g^{b}[p-1,p-1]}\;. \qquad (36)$$

From Equations (26) and (28), we can find the prediction error variance

$$\begin{aligned} g^{f}[p,0] &= R_{e^{f}[p]\,x}[0] = \sigma^{2}[p]\;, \\ g^{b}[p,p] &= R_{e^{b}[p]\,x}[p] = \sigma^{2}[p]\;. \end{aligned} \qquad (37)$$

We summarize the Schur algorithm as follows:

STEP 0. $g^{f}[0,k] = g^{b}[0,k] = R_{x}[k] \qquad k = 0,1,...,M$

STEP 1. $p = 0$

STEP 2. $$k[p+1] = \frac{g^{f}[p,p+1]}{g^{b}[p,p]} \qquad (38)$$

STEP 3. $g^f[p+1,k] = g^f[p,k] - k[p+1]\, g^b[p,k-1]$ $\quad k = p+1, p+2, ..., M$ (39)

STEP 4. $g^b[p+1,k] = g^b[p,k-1] - k[p+1]\, g^f[p,k]$ $\quad k = p+1, p+2, ..., M$ (40)

STEP 5. **Go to step 1 and increase the order by 1**

STEP 6. At the last order of M: Prediction Error Variance

$$\sigma_e^2 = g^b[M,M] \tag{41}$$

The Schur algorithm computes the reflection coefficients as the ratio of two gapped functions, and most importantly, computations in steps 3 and 4 can be executed in parallel.

F. SPLIT SCHUR ALGORITHM

It can be shown that using a symmetric vector and computing only half of the terms, computation complexity can be reduced 50% in Schur algorithm. Since the new variables in split Schur algorithm are not related to basic linear prediction, only the algorithm will be presented here. Details of derivation of the parameters used in this algorithm can be found in [Ref. 10].

The split Schur algorithm can be summarized as follows:

STEP 0. $k[0] = 0 \quad ; \quad g[0,0] = R_x[0]$

$$g[0,k] = 2\, R_x[k] \quad ; \quad g[1,k] = R_x[k] + R_x[k-1] \qquad k = 1, 2, ..., M$$

STEP 1. $p=0$

STEP 2. $$\alpha = \frac{g[p+1,p+1]}{g[p,p]} \tag{42}$$

STEP 3. $$k[p+1] = -1 + \frac{\alpha}{1 - k[p]} \quad (43)$$

STEP 4. $$g[p+2,k] = g[p+1,k] + g[p+1,k-1] - \alpha \; g[p,k-1]$$

$$k = p+2, p+3, ..., M \quad (44)$$

STEP 5. **Go to step 1 and increase the order by 1**

STEP 6. At the final order of M: $$\sigma_e^2 = g[M,M]\,(1 - k[M]) \quad (45)$$

We note that step 4 requires only one multiplication for each order whereas step 3 and 4 in Schur algorithm requires total two multiplications which reduces the computation by 50%.

G. AR SPECTRAL ESTIMATION

An estimate of the Power Spectral Density (PSD) of a process can be obtained by using the prediction coefficients and the variance obtained from the AR algorithms. Once the AR parameters have been obtained, the PSD of the model of order M can be determined by the equation [Ref. 7],

$$P_{AR}(f) = \frac{\sigma_e^2}{\left| 1 + \sum_{k=1}^{M} a[M,k] \exp(-j2\pi fk) \right|^2} . \quad (46)$$

The above PSD estimation is not the same as the true PSD obtained from the infinite autocorrelation lags. Since an assumption has been made that the signal is an AR (M) process, Eq. (46) is the best estimation, and is only valid for Gaussian random processes and known autocorrelation lags [Ref. 7]. In practice, if the Burg method is used for finding the model parameters, then the spectral estimation is called the Maximum Entropy Spectral Estimation [Ref. 10].

It is always desired to extract the signal from the noise. The autoregressive estimation is one of the methods commonly used in the DSP applications. The algorithms discussed in previous sections provide fast methods to obtain the estimation parameters to model the signal from the given input data.

V. TEST PROCEDURES AND RESULTS

The main goal of this work is to develop a library file that contains the AR routines to implement the AR algorithms discussed in Chapter IV. The first section in this chapter describes the subroutines implemented in this work. The second section lists the batch files that have been used to obtain the object files, to create the library, and to check the results of the subroutines by using the PC and the C30 programs. The results in terms of the cycle counts to execute the subroutines are discussed in the last section.

A. SUBROUTINES

The subroutines implemented in this work are listed in Table 5-1. All the AR routines have been written in C language. In addition, the subroutine that computes the autocorrelation function XCOR has been written in assembly code. Using the C30 compiler and library built utilities, a library file *dsplib.lib* was created. The C language codes for the subroutines and the batch files to obtain the object files and to create the library are listed in Appendix A. Appendix B contains the assembly language code for the subroutine XCOR, and the batch files to obtain the object files and to create the library are listed in Appendix C.

Since the speed is the most important factor in real time applications, the following adjustments have been made in the subroutines and the main C30 program to speed up the execution. These adjustments simplify the subroutines by avoiding unnecessary loops. Here, loop refers to the process to find the AR parameters for each order from k=1 to k=P, where P is the final prediction filter order.

1. In all subroutines but XCOR and SPECTRA, the first leg of the loop (k=1) has been done prior to the loop, so the loop starts from k=2. In addition to this, the loop starts from k=3 for the subroutines LEVPRE and LEVREF, and from k=4 for the subroutine RCTOPC. For the subroutines SCHUR and SPSCHUR, the last leg of the loop (k=P) is executed outside the loop.

2. The length of the arrays in the main program that return the results of the reflection or prediction coefficients must be P+1 for all subroutines that compute the variance, because

the variance is placed as the last element in the array. The first P elements in the array correspond to the prediction or reflection coefficients, the (P+1)st element is the prediction error variance.

Table 5-1. List of AR Routines

SUBROUTINE NAME	DESCRIPTION
XCOR	Calculates the autocorrelation lag values of a given input data.
BURG	Calculates the reflection coefficients and prediction error variance of the given order from the given input data using Burg algorithm.
LEVPRE	Calculates the prediction coefficients and prediction error variance of the given order from the given autocorrelation values using the Levinson algorithm.
LEVREF	Calculates the reflection coefficients and prediction error variance of the given order from the given autocorrelation values using the Levinson algorithm.
RCTOPC	Calculates the prediction coefficients from the given reflection coefficients using the Levinson algorithm.
SCHUR	Calculates the reflection coefficients and prediction error variance of the given order from the given autocorrelation values using the Schur algorithm.
SPSCHUR	Calculates the reflection coefficients and prediction error variance of the given order from the given autocorrelation values using the Split Schur algorithm.
SPECTRA	Calculates the power spectral density of a model from the given prediction coefficients and prediction error variance .

3. Memory allocation for the local arrays in the subroutines takes considerable amount of time. In order to decrease the execution time, the C30 program for the subroutines LEVPRE, LEVREF, and RCTOPC allocates larger arrays for the variables that return the data from the C30. For example, the arrays "pc" and "rc" return an array with (P*(P+1)/2)+1 elements. The first P elements are the reflection or prediction coefficients, the (P+1)st element

for the subroutines LEVPRE and LEVREF is the prediction error variance. The rest of the elements in the arrays are ignored.

B. BATCH FILES

The batch files are small programs that contain the commands to compile, assemble and link in a single step. Typing the name of the batch file executes all the commands in the batch file. The following subsections describe the batch files that have been used to obtain the object files, to create the library, and to check the results of the subroutines by using the PC and the C30 programs. All the batch files are listed in Appendix C.

1. Obtaining Object Files of Subroutine Codes

The following batch file invokes the C30 compiler to obtain the executable object files by compiling and assembling the C codes in a single pass. The -o option enables the optimizer of the C30 compiler.

MKOPT.BAT

cl30 -o -s -al xcor.c burg.c levpre.c levref.c rctopc.c schur.c spschur.c spectra.c

-o: is the optimization level. It can be -o0, -o1, or -o2 (or -o). If -o option is not used, then the code is not optimized. The C compiler optimizer improves the execution speed by simplifiying loops, rearranging the statements and expressions and allocating variables into registers depending on the optimization level. [Ref. 3]

-s: causes the compiler to produce the assembly language code of the C language program 30xcor.c with the name 30xcor.asm.

-al: causes the assembler to produce a listing file 30xcor.lst in assembly language. The corresponding C program codes are placed as comments in .lst file which is used for debugging.

2. Creating Library

The C30 archiver allows us to collect the individual object files into a single library file. The following batch file invokes the C30 archiver and puts the optimized object files obtained by the previous batch file into a library file, dsplib.lib.

MKLIB.BAT

```
ar30 -r dsplib.lib xcor.obj burg.obj levpre.obj levref.obj rctopc.obj schur.obj spschur.obj spectra.obj
```

3. Compiling and Linking C30 and PC Programs

As described in Chapter III, we can transfer the data back and forth between the C30 board and the PC. This batch file can be used to compile, assemble and link the C30 and PC programs to check the result of the subroutines. The following batch file has been written specially for the programs listed in Appendix D.

MKXCOR.BAT

```
cl30 -s -al 30xcor.c -z -cr -m 30xcor.map lsicmap.cmd lsiboot.obj -l dsplib.lib rts30b.lib -o 30xcor.out
bcc -ml -P -CP -I c:\c30;c:\cpp\lib -i c:\30;c:\cpp\include -erxcor rxcor.cp lm30devlib
```

The first line (cl30) compiles the C30 C code 30XCOR.C and links it with the libraries. The options are as follows.

-z: causes the rest of the commands to pass to the linker.

-cr: causes the linker to use the RAM model. It is essential to use the "coffload()" command in the PC program to load the C30 program when using -cr option.

-m 30xcor.map: causes the linker to produce an output map file with the name 30xcor.map which shows the memory locations that the linker has placed the code and the variables.

The file lsicmap.cmd is a linker command file that tells the linker where to map the program in the memory of the C30 board. It also includes the memory locations of the interrupt vectors and the flags (Comm0-Comm15) that are used in data transfer between the

C30 board and the PC. Appendix F contains the LSICMAP.CMD memory map file used in this work.

The file lsiboot.obj contains the code and data to initialize the C30 processor and calls the main () function. It has been written specially for the LSI C30 board, and it must be listed before the rts30b.lib. When the program starts running, it executes the lsiboot.obj first.

-l dsplib.lib rts30b.lib causes the linker to use the libraries dsplib and rts30b. The library dsplib.lib contains the object files obtained from the AR routine codes. The function call "xcor" in the C30 program requires the linker to link the dsplib library. The library rts30b.lib contains the object files of the runtime support functions obtained from the source file rts.src. The option -o 30xcor.out causes the linker to name the output object file as 30xcor.out which is used in the PC program.

The second line (bcc) contains the commands to compile the PC C code with Borland C compiler and link them with the LSI interface library lm30dev.lib. The option -ml causes the compiler to use the large memory model, -e option names the output executable object file as rxcor, and rxcor.cp is the name of the PC C program.

4. Adding New Routines To Library

The routines in the library can be modified to decrease the execution time or new routines can be added to the library. This can be done easily by executing the following steps:

1. If the new or modified function, newfunc, has been written in C code, then the following batch file is used to obtain the object file. The optimization or other options described in previous subsections are optional.

MKNEWFUN.BAT

cl30 -o newfunc.c

2. It is also possible that after using cl30 command shell with -s option to obtain the assembly code, modification can be done on the assembly language code. Then, new assembly code is reassembled to obtain the object file.

3. If the function has been written in assembly code, then the following batch file is used to obtain the object file.

MKNEWFUN.BAT

asm30 newfunc.asm newfunc.obj

4. After having the object file, following batch file is used to add the new file to the library.

MKNEWFUN.BAT

ar30 -r dsplib.lib newfunc.obj

C. RESULTS

The major concern in creating the library is to keep the execution time for each subroutine as low as possible. One of the factors used to decrease the execution time is the optimizer of the C30 compiler. Enabling the C30 optimizer (-o option) reduces the execution time by simplifying the C code depending on the optimization level. The other factor is the 64 words instruction cache. The instruction cache stores often repeated sections of code, thus reducing the number of off-chip accesses. The instruction cache is enabled by writing 1 to the 11th bit in the status register (ST) by using inline assembly language code in the main C30 program. The following code in the main program enables the cache.

```
asm("    OR    800h, ST");
```

We know that while an instruction is being executed, the next three instruction are being consequently read, decoded, and fetched, thus improving the performance of the C30 processor. But this may not be the case every time. The three conflicts, pipeline, register, and memory conflicts, cause delays or instructions to be executed in many cycles than it is expected. For example, during the execution of instructions, such as BR, DB, RPTS, and RPTB, the pipeline becomes inactive causing delay; if an auxiliary register (AR) is loaded and a different auxiliary register is used on the next instruction, decoding of the second instruction

is delayed two cycles (60 ns cycle); most of the instructions requiring an access (write or read) to an external memory area are executed in two cycles [Ref. 1]. These conflicts increase the execution time.

The functional correctness of the subroutines has been tested using the input data saved in a PC file. The data has been transferred to the external dual access memory area (30000h) on the C30 board. The result computed by the C30 was sent back to the PC for display.

Two approaches have been followed in this work to find the execution time. The first one is using Timer 0 in clock mode. The value in the counter register has been read right before and after the function call. The difference between the two values in the counter register is the execution time of that particular function after multiplied by 120 ns, since the value in the counter register is incremented once in every 120 ns. This is illustrated in the C30 program in Appendix D.

The numbers in Table 5-2 show the cycle counts for each subroutine without the optimization, with the highest optimization (-o) and with the cache enabled while -o option is in use. These numbers have been found using Timer 0 in clock mode (each cycle is 120 ns), and using the external dual access memory area of the C30. Note that the subroutine SPECTRA computes the estimate of the PSD for the positive frequencies by using 32 points.

In many applications where the execution time is critical, it may be necessary to write functions in assembly language code. The number of cycles required to execute the assembly language code for the subroutine XCOR listed in Appendix B is 3664 (and 3607 with cache enabled), 34 cycles less than the optimized C code. This reduces the execution time by 120 x 34 ns (4.08 microseconds).

Table 5-2. Cycle Numbers For AR Routines (N = 160, P = 10)

SUBROUTINE NAME	CYCLE COUNTS without optimization	CYCLE COUNTS with highest optimization	CYCLE COUNTS cache enabled
XCOR	23484	3698	3641
BURG	61595	15536	12391
LEVPRE	2228	987	923
LEVREF	1968	925	883
RCTOPC	1210	512	412
SCHUR	1904	983	923
SPSCHUR	2313	1257	1211
SPECTRA	45083	41280	41219

The second way to find the total execution time of a function is to count the cycles of each instruction from the assembly language code. Since the cycles required to execute each instruction are known, we can count the cycles manually for each instruction by taking into account the loops. Note that here each cycle is 60 ns. But this is the ideal case where there is no pipeline, register or memory conflicts.

The cycle counts obtained from the assembly code are listed in Table 5-3 for N=160 and P=10. Since some of the subroutines use the functions *malloc, free, sin, cos*, and *log10*, and there is no accurate information about the cycle counts to execute these functions. We know from the rts.src source file that inverse of a number takes 36 cycles, but there is no information about the other functions. Texas Instruments reports that sin and cos functions take 22-23 cycles and log10 takes 27-40 cycles [Ref. 3]. We estimate that malloc and free functions take 45-50 cycles.

Table 5-3. Cycle Counts Obtained From Assembly Language Code

SUBROUTINE NAME	TMS320C30	DSP561xx (Note *)
XCOR	2003	2282
BURG	15859	13620
LEVPRE	1157	1309
LEVREF	1168	1306
RCTOPC	410	492
SCHUR	1347	619
SPSCHUR	1519	*
SPECTRA	23614	*

Note *: There is no information about those two functions in Ref. 11.

Table 5-3 also compares the cycle counts of the TMS320C30 and Motorola DSP561xx digital signal processors. Since there is not enough information about how these numbers were found [Ref. 11], it is assumed that these numbers were determined by counting the instruction cycles. The function prototypes are also not known, i.e., whether the variance is returned separately or as the last element after the AR parameters. The purpose of the comparison is to show the difference of the count cycles between the two processors, not to say that one of the processors is better than the other.

The execution time for different data lengths and order sizes using the external dual access memory area on the C30 board with highest optimization level and cache enabled are listed in Table 5-4.

Table 5-4. Count Cycles For Different Data Lengths N and Order Sizes P

SUBROUTINE NAME	N=160 P = 10	N=256 P = 10	N=256 P = 15	N=512 P = 15	N=512 P = 20
XCOR	3641	5789	8314	16644	21722
BURG	12391	19489	28320	56230	73849
LEVPRE	923	923	1626	1626	2504
LEVREF	883	883	1584	1584	2446
RCTOPC	412	412	777	777	1252
SCHUR	923	923	1466	1466	2145
SPSCHUR	1211	1211	1965	1965	2896
SPECTRA	41219	41219	60777	60777	80480

The functional correctness of the AR routines has been tested successfully using the PC, the C30 board, and the library file dsplib.lib. Since the data transfer between the PC and the C30 board are performed via the dual access external memory area (30000h), the numbers in Table 5-2 are higher than the numbers in Table 5-3 obtained directly by counting the instruction cycles from the assembly language codes. The cycle counts for different data lengths and order sizes are also listed in Table 5-4.

If the library DSPLIB.LIB is desired to be used in some applications with a PC, then one should

1. write a C source code for the C30 (.C) which must include the header file DSP.H,
2. write a PC C source code (.CP),
3. modify the batch file XCOR1.BAT in Appendix C with new names.

The library file DSPLIB.LIB should be linked, and the C30 source code should include the header file DSP.H every time these AR routines are used.

VI. CONCLUSIONS AND RECOMMENDATIONS

Autoregressive analysis is one of the methods commonly used in DSP applications for modeling and estimation of random signals. The Levinson, Burg and Schur algorithms, discussed in Chapter IV, provide fast methods to find the prediction and reflection coefficients and prediction error variance to model the signal from the given input data. High speed digital signal processors with specialized instruction sets are used in these applications to carry out desired computations in realtime; the Texas Instruments' TMS320C30 digital signal processor on the spectrum TMS320C30 System Board has been used in this work.

The TMS320C30 digital signal processor has 16 million words of memory space and 60 ns single cycle execution time or a 16.7 MIPS instruction rate. It is a 32-bit processor and supports both fixed and floating point operations. The advanced architecture, rich instruction set, and high speed make the TMS320C30 digital signal processor very suitable to implement the DSP applications in real time for up to 152 KHz sampling rate.

The C30 has a powerful C compiler and development tools to compile and link C language and assembly language programs. The compilation process is easy and performs multiple compilations in a single step. The interface library allows the programmer to use the PC and the C30 board together to perform the DSP applications. Analog interfaces on the chip and interrupts enhance the capability of the C30 board to input/output the data from/to outside sources.

This work created a library file that contains the object files of AR routines to compute the prediction and reflection coefficients and the prediction error variance from the given input data. The assembly language and C code programs and batch files listed in Appendices give an idea on how to use the PC and the C30 board together to implement the DSP applications.

Although the split Schur algorithm theoretically decreases the computation cost almost 50% compared to the Schur algorithm, we could not obtain this result. As we see from Table 5-4, the split Schur algorithm takes more time than the Schur algorithm.

One of the interesting observations from Table 5-4 is that the Schur algorithm is executed in less time than the Levinson (LEVREF) algorithm as the prediction filter order P increases even though both algorithms use the autocorrelation values as input and compute the reflection coefficients and the variance.

Table 5-2 shows how we can improve the execution time by using the C30 compiler optimizer. Enabling the C30 cache also helps to decrease the execution time by storing often repeated codes in its cache memory. The difference between the numbers without the optimization and with the optimization and cache enabled are quite noticeable.

The functional correctness of the AR routines has been tested successfully using the PC, the C30 board, and the library file DSPLIB.LIB. Since the data transfer between the PC and the C30 board are performed via the dual access external memory area (30000h), and by taking into account the pipeline and register conflicts, the numbers in Table 5-2 are higher than the numbers in Table 5-3 obtained directly by counting the instruction cycles from the assembly language codes.

The AR routines in this library could be improved to decrease the execution time. The use of a simulator to determine conflicts and more accurately estimate execution time would be a good first step. Hand optimization of the assembly language code using the information from the simulator could reduce execution times by up to 50%.

APPENDIX A: TMS320C30 SOURCE CODE FOR AR ROUTINES

This appendix contains the C language source code of the AR routines discussed in Chapter IV. The header file DSP.H that includes the function prototypes of the routines is listed at the end of the appendix.

XCOR.C

```
#include "dsp.h"

/*********************************************************************
/  void xcor (float *x, float *Rx, int N, int P)
/
/  Calculates the normalized autocorrelation lag values of a given sequence
/
/  x: input data x(0), x(1), ..., x(N-1)
/  Rx: autocorrelation values of input data such as R(0),R(1),...R(P)
/  N: length of the data
/  P: length of the filter; number of autocorrelation lag values.
*********************************************************************/

void xcor(float *x,float *Rx,int N,int P)
{
        int m,n,k;
        float z;
                                        /* Implements Eq. (12) in Chapter IV*/
        for (m=0; m<=P; m++) {
                z = 0.0;
                k = N - m;

                for (n=0; n<k; n++)             /* Inner loop in Eq. (12)          */
                        z += *(x+n+m) * *(x+n);

                *Rx++ = z / N;                  /* Normalization step              */
        }
}
```

BURG.C

```
#include "dsp.h"

/*************************************************************************
/  void burg (float *x, float *bpc, int P, int NM1)
/
/ Calculates the reflection coefficients of the given order and the prediction error variance
/  from the given input data using the Burg algorithm.
/
/  x: input data x(0), x(1), ..., x(N-1)
/  bpc: reflection coefficients (rc)  and the variance (var)  (rc1, rc2, ..., rcP, var)
/  P: length of the filter; order size
/  NM1: N (data length) - 1
/*************************************************************************/

void burg(float *x,float *bpc,int P,int NM1)
{
        int k,i,m,n;
        float num,den,*ef,temp,gamma,var,temp1,temp2,temp3,temp4;

        n=NM1+1;                                /* n is the data length                 */
        ef = (float *) malloc(2*n*sizeof(float));   /* Memory allocation for prediction error*/
        var = 0.0;

        for (i=0; i<=NM1; i++) {                /* Initialization  (Step 0) in Burg algorithm*/
                var = var + (*x * *x);          /* var = R[0]                           */
                *(ef+n+i) = *x;                 /* Backward (bw) prediction error       */
                *(ef+i) = *x++;                 /* Forward (fw) prediction error        */
        }
        var = var / n;

/* Loop for order size of k = 1                                                        */

        num = 0.0;                              /* Initialization for numerator in Eq. (22)*/
        den = 0.0;                              /* Initialization for denominator in Eq. (22)*/
        for (m=1; m<=NM1; m++) {
                num += *(ef+m) * *(ef+n+m-1); /* Computes the numerator in Eq. (22)*/
                den += *(ef+m) * *(ef+m);     /* Computes the denominator in Eq. (22)*/
                den += *(ef+n+m-1) * *(ef+n+m-1); /*            "                    */
        }
        gamma =  2.0 * num;
        gamma = gamma / den;                    /* Eq. (22) , Step 2                    */
```

```
        *bpc++ = gamma;                 /* Reflection coefficient, k[1]        */
        temp = 1.0 - gamma * gamma;     /* Eq. (24), Step 4                    */
        var = var * temp;               /* Variance, Eq. (24), Step 4          */

        for (i=NM1; i>=1; i--) {        /* Updates the fw. and bw. prediction errors*/
                temp3 = *(ef+i);        /* Step 5 in Eq. (25a) and Eq. (25b)   */
                temp4 = *(ef+n+i-1);
                *(ef+i) = temp3 - gamma * temp4;    /* Eq. (25a)               */
                *(ef+n+i) = temp4 - gamma * temp3;/* Eq. (25b)                 */
        }

/* Loop for order size k = 2, 3, ..., P                                        */

        for (k=2; k<=P; k++) {
                num = 0.0;
                for (m=k; m<=NM1; m++)
                        num += *(ef+m) * *(ef+n+m-1);

                temp1 = *(ef+k-1) * *(ef+k-1);
                temp2 = *(ef+n+NM1) * *(ef+n+NM1);
                den = temp * den - temp1 - temp2;   /* Eq. (21) computes the denominator*/
                gamma =  2.0 * num;                 /* recursively              */
                gamma = gamma / den;        /* Eq. (22)                        */
                *bpc++ = gamma;             /* Reflection coefficients, k[k]   */
                temp = 1.0 - gamma * gamma;
                var = var * temp;           /* Eq. (24)                        */

                if ( k<P ) {                        /* Updating the fw. and bw. prediction*/
                        for (i=NM1; i>=k; i--) {    /* errors in Eq. (25a) and Eq. (25b)*/
                                temp3 = *(ef+i);    /* is not necessary for the last order */
                                temp4 = *(ef+n+i-1); /* size k = P             */
                                *(ef+i) = temp3 - gamma * temp4;
                                *(ef+n+i) = temp4 - gamma * temp3;
                        }
                }
        }
        *bpc++ = var;           /*Variance is returned back from the subroutine as the last*/
                                /* element of the array that holds the reflection coefficients*/
        free(ef);
}
```

LEVPRE.C

```
#include "dsp.h"

/*************************************************************************
/  void levpre (float *ac, float *pc, int P, int size)
/
/  Calculates the linear prediction coefficients of the given order and the prediction error
/  variance from the given input autocorrelation values using the Levinson algorithm.
/
/  ac: autocorrelation values R(0), R(1), ..., R(P)
/  pc: prediction coefficients (pc) and the variance (pc1, pc2, ..., pcP, var). The length of pc
/       array is size. But the elements after the var, (P+1)st element, are ignored, not used.
/  P: order size
/  size: P * ( P +1 ) / 2 + 1
/*************************************************************************/

void levpre (float *ac,float *pc, int P,int size)
{
        int i,n,k,c,m;
        float g,var,gamma;

/* Loop for order size k = 1                                                    */
                                        /* Step 0 in Levinson algorithm        */
        *(pc+1) = -(*(ac+1)) / *ac;
        var = 1.0 - *(pc+1) * *(pc+1);
        var = var * *ac;                /* Variance                            */

/* Loop for order size k = 2                                                    */

        g = *(pc+1) * *(ac+1);          /* Eq. (14), Step 2 in Levinson algorithm*/
        g = g + *(ac+2);
        *(pc+3) = - g / var;
        gamma = *(pc+3);
        var = var * (1.0 - gamma * gamma) ; /* Eq. (16), Step 4                */
        *(pc+2) = *(pc+1) + gamma * *(pc+1);    /* Eq. (15), Step 3            */

        c=2;

/* Loop for order size k = 3, 4, ..., P                                         */

        for (k=2; k<P; k++) {
```

```
            g = 0.0;
            for (n=0; n<k; n++)
                  g = g + *(pc+c+n) * *(ac+k-n);/* Numerator in Eq. (14)     */

            g = g + *(ac+k+1);
            c = c+k;
            *(pc+c+k) = - g / var;                  /* Eq. (14)                */
            gamma = *(pc+c+k);
            var = var * (1.0 - gamma * gamma) ; /* Eq. (16)                    */

            for (i=0; i<k; i++)                     /* Eq. (15)                */
                  *(pc+c+i) = *(pc+c-k+i) + gamma * *(pc+c-1-i);
      }

/* Prediction parameters are the pcs that were computed at the last order      */

      for (i=1; i<=P; i++)
            *(pc+P-i) = *(pc+size-i);
      *(pc+P) = var;          /*Variance is returned back from the subroutine as the last*/
                              /* element of the array that holds the reflection coefficients*/
}
```

LEVREF.C

```
#include "dsp.h"

/*************************************************************************
/  void levref (float *ac, float *rc, int PM1)
/
/  Calculates the reflection coefficients of the given order and the prediction error variance
/  from the given input autocorrelation values using the Levinson algorithm.
/
/  ac: autocorrelation function R(0), R(1), ..., R(P)
/  rc: reflection coefficients (rc) and the variance (rc1, rc2, ..., rcP, var). The length of rc
/      array is size. But the elements after the var, (P+1)st element, are ignored, not used.
/  PM1: P (order size) - 1
*************************************************************************/

void levref (float *ac, float *rc, int PM1)
{
        int i,n,k,c,p;
        float g,var,gamma;

/* Loop for order size k = 1                                                    */

        *(rc+1) = -(*(ac+1)) / *ac;          /* Step 0 in Levinson algorithm    */
        *rc = *(rc+1);                       /* Reflection coefficient, k[1]    */
        var = 1.0 - *(rc+1) * *(rc+1);       /* Variance                        */
        var = var * *ac;                     /*    "                            */

/* Loop for order size k = 2                                                    */

        g = *(rc+1) * *(ac+1);               /* Eq. (14), Step 2                */
        g = g + *(ac+2);                     /*      "                          */
        *(rc+3) = - g / var;                 /* Reflection coefficient, k[2]    */
        gamma = *(rc+3);
        var = var * (1.0 - gamma * gamma) ; /* Variance, Eq. (16), Step 4       */

        *(rc+2) = *(rc+1) + gamma * *(rc+1);        /* Eq.(15), Step 3          */
        *(rc+1) = gamma;                     /* Reflection coefficient, k[2]    */
        c=2;
```

```
/* Loop for order size k = 3, 4, ..., P                                    */

        for (k=2; k<=PM1; k++) {

                g = 0.0;
                for (n=0; n<k; n++)
                        g = g + *(rc+c+n) * *(ac+k-n);      /* Numerator in Eq. (14)*/

                g = g + *(ac+k+1);              /* Numerator in Eq. (14)       */
                c=c+k;
                *(rc+c+k) = - g / var;          /* Eq. (14)                    */
                gamma = *(rc+c+k);              /* Reflection coefficient, k[k]*/
                var = var * (1.0 - gamma * gamma) ;     /* Eq. (16)            */

                if ( k < PM1 ) {                /* It is not necessary to execute Eq. (15)*/
                        for (i=0; i<k; i++)     /* at the last order           */
                                *(rc+c+i) = *(rc+c-k+i) + gamma * *(rc+c-1-i);/* Eq. (15)*/
                }
                *(rc+k) = gamma;                /* Reflection coefficient, k[k]*/
        }
        p=PM1+1;

        *(rc+p)= var;           /*Variance is returned back from the subroutine as the last*/
                                /* element of the array that holds the reflection coefficients*/

}
```

RCTOPC.C

```
#include "dsp.h"

/*************************************************************************
/  void rctopc (float *rcof, float *pc, int P, int size)
/
/  Calculates the linear prediction coefficients from the given reflection coefficients.
/
/  rcof: reflection coefficients (rc1, rc2, ..., rcP)
/  pc: prediction coefficients (pc1, pc2, ..., pcP). The length of pc array is size. But the /
   elements after the var, (P+1)st element, are ignored, not used.
/  P: order size
/  size: P * ( P + 1 ) / 2 + 1
*************************************************************************/

void rctopc (float *rcof,float *pc,int P,int size)
{
        int i,k,c,m;
        float temp;

/* Loop for order size k = 1                                              */

        *(pc+1)= *rcof++;                       /* Reflection coefficient, k[1] */

/* Loop for order size k = 2                                              */

        *(pc+3) = *rcof;                        /* Reflection coefficient, k[2] */
        *(pc+2) = *(pc+1) + *rcof++ * *(pc+1);  /* Eq. (15)                     */

/* Loop for order size k = 3                                              */

        *(pc+6) = *rcof;                        /* Reflection coefficient, k[3] */
        temp = *rcof++;

        *(pc+4)= *(pc+2) + temp * *(pc+3);      /* Eq. (15)                     */
        *(pc+5)= *(pc+3) + temp * *(pc+2);      /*      "                       */
        c=4;

/* Loop for order size k = 4, 5, ..., P                                   */

        for (k=3;k<P;k++) {
                c = c + k;
```

```
            *(pc+c+k) = *rcof;                  /* Reflection coefficient, k[k] */
            temp = *rcof++;

            for (i=0;i<k;i++)                   /* Eq. (15)                     */
                  *(pc+c+i)= *(pc+c-k+i) + temp * *(pc+c-1-i);
      }

/* Prediction parameters are the pcs that were computed at the last order       */

      for (i=1; i<=P; i++)
            *(pc+P-i) = *(pc+size-i);
}
```

SCHUR.C

```
#include "dsp.h"

/*************************************************************************
/  void schur(float *Rx, float *src,int P, int PM1)
/
/  Calculates the reflection coefficients of the given order and the prediction error variance
/  from the given input autocorrelation values using the Schur algorithm.
/
/  Rx: autocorrelation values R(0), R(1), ..., R(P)
/  src: reflection coefficients (rc) and the variance  (rc1, rc2, .., rcP, var)
/  P: order size
/  PM1: P - 1
*************************************************************************/

void schur(float *Rx,float *src,int P,int PM1)
{
        int n,h,m,tempi;
        float *pt,*ptp,*en,ex,var,gamma,temp;

        en = (float *) malloc(2*P*sizeof(float));       /* Memory allocation for bw. and fw.*/
        pt = en + P;                                    /* prediction errors. "pt" is the last P */
        ptp = en + PM1;                                 /* location corresponding to the fw. pre.errors*/
        for (n=0; n<=PM1; n++) {
                *(en+n) = *Rx++;                        /* Initialization; bw. prediction errors*/
                *(pt+n) = *Rx;                          /* Initialization; fw. prediction errors*/
        }

/* Loop for order size k = 1                                                    */

        var = *en;                                      /* Variance             */
        temp = *(pt+PM1);
        gamma = - *pt / var;
        *src++ = gamma;                                 /* Reflection coefficient, k[1]  */
        var += gamma * *pt;                             /* Variance             */

        temp = temp + gamma * *ptp;
        for (m=1; m<PM1 ;m++) {                         /* Updating the bw. and fw.      */
                ex = *(pt+m) + gamma * *(en+m);         /* prediction errors            */
                *(en+m) += gamma * *(pt+m);
                *(pt+m) = ex;
        }
```

```
/* Loop for order size k = 2, 3, ..., P-1                                    */

        for (h=1;h<PM1;h++) {
                gamma = - *(pt+h) / var;
                *src++ = gamma;             /* Reflection coefficient, k[k]   */
                var += gamma * *(pt+h);     /* Variance                       */

                temp = temp + gamma * *(ptp-h);
                n = h+1;
                tempi = 1;
                for (m=n; m<PM1 ;m++) {                 /* Updating bw. and fw.*/
                        ex = *(pt+m) + gamma * *(en+tempi);/* prediction errors */
                        *(en+tempi) += gamma * *(pt+m);
                        *(pt+m) = ex;
                        tempi++;
                }
        }

/* Loop for order size k = P                                                 */

        gamma = - temp / var;       /* Variance                              */
        *src++ = gamma;             /* Reflection coefficient, k[P]          */
        var += gamma * temp;        /* Variance                              */
        *src++ = var;           /*Variance is returned back from the subroutine as the last*/
                                /* element of the array that holds the reflection coefficients*/
        free(en);
}
```

SPSCHUR.C

```
#include "dsp.h"

/*************************************************************************
/  void spschur(float *Rx, float *src, int P)
/
/  Calculates the reflection coefficients of the given order and the prediction error variance
/  from the given input autocorrelation values using the Split Schur algorithm.
/
/  Rx: autocorrelation values R(0), R(1), ..., R(P)
/  src: reflection coefficients (rc) and the variance (rc1, rc2, ..., rcP, var)
/  P: order size
*************************************************************************/

void spschur (float *Rx, float *src, int P)
{
        int m, n;
        int k,i,L;
        float alpha,*vp,*pt,gamma,temp,nn;

        vp=(float *) malloc(2*P*sizeof(float));      /* Memory allocation          */

        pt = vp + P;                                 /* Initialization Step 0 in Split Schur */
        *pt = *Rx;                                   /* algorithm                  */
        *vp  = *Rx + *(Rx+1);                        /*          "                 */

        for (i=1;i<P;i++) {                          /*          "                 */
                *(pt+i) = 2 * *(Rx+i);               /*          "                 */
                *(vp+i) = *(Rx+i) + *(Rx+i+1);/*          "                 */
        }
        L = P;

 /* Loop for order size k = 1                                                      */

        alpha = *vp / *pt;                           /* Eq. (42), Step 2           */
        gamma = 1.0 - alpha;                         /* Eq. (43), Step 3           */
        L--;
        *src++ = gamma;                              /* Reflection coefficient, k[1] */
        for (n=0;n<L;n++) {                          /* Updating, Eq. (44), Step 4 */
                *pt++ = *(vp+n);
                nn = *(vp+n) + *(vp+n+1);
                *(vp+n) = nn - alpha * *pt;
```

```
        }

/* Loop for order size k = 2, 3, ..., P-1                                  */

        for (k=2;k<P;k++) {
                pt = vp+P;
                alpha = *vp / *pt;          /* Eq. (42)                    */
                temp = 1.0 + gamma;         /* Eq. (43)                    */
                gamma = alpha / temp;       /*      "                      */
                gamma = 1.0 - gamma;        /*      "                      */
                *src++ = gamma;             /* Reflection coefficient, k[k]*/
                L--;
                for (n=0;n<L;n++) {         /* Updating, Eq. (44)          */
                        *pt++ = *(vp+n);
                        nn = *(vp+n) + *(vp+n+1);
                        *(vp+n) = nn - alpha * *pt;
                }
        }

/* Loop for order size k = P                                               */

        pt = vp+P;
        alpha = *vp / *pt;                  /* Eq. (42)                    */
        temp = 1.0 + gamma;                 /* Eq. (43)                    */
        gamma = alpha / temp;               /*      "                      */
        gamma = 1.0 - gamma;                /*      "                      */
        *src++ = gamma;                     /* Reflection coefficient, k[P]*/
        temp = 1.0 + gamma;                 /* Variance, Eq. (45), is returned back */
        *src++ = temp * *vp;                /* from the subroutine as the last element*/
                                            /* of the array that holds the reflection*/
                                            /* coefficients                */

        free(vp);
}
```

SPECTRA.C

```
#include "dsp.h"
#include <c:\tms_tool\math.h>

/*************************************************************************
/  void spectra (float *pc,float *arpsdlog,int P, int W, float freqres)
/
/  Calculates the estimates of the power spectral density coeefficients of an input using
/  the linear prediction coefficients and the variance obtained from the Levinson algorithm.
/
/  pc: prediction coefficients (pc) and the variance (pc1, pc2, ..., pcP, var)
/  arpsdlog : logarithmic power spectral density (PSD) coefficients. The size of this array is
/             (freqres + 1).
/  P: order size
/  W: 2*pi
/  freqres: number of the PSD coefficients for the positive frequencies (0 - 0.5*fs)  minus 1.
/           For example; to have 32 points between 0 and 0.5fs, freqres must be 31.
*************************************************************************/

void spectra (float *pc,float *arpsdlog,int P, int W, float freqres)
{
        float f,w,den,power;
        int q,k,kk;
        COMPLEX  e;

        q=0;

/* Only the positive frequencies are computed                                   */

        for (f=0.0; f<0.5; f += freqres) {
                e.real = 1.0;                           /* Exponention term is divided*/
                e.imag = 0.0;                           /* into its sin and cos terms*/
                w = W * f;                              /* 2 * pi * f           */

/* First P elements in the pc array are the prediction coefficients and the last element is the
variance.                                                                    */

                for (k=0; k<P; k++) {
                        kk = k + 1;
                        e.real += pc[k] * cos(w * kk);          /* exp(-j*2*pi*f*k)    */
                        e.imag += pc[k] * sin(w * kk);          /*       "             */
                }
```

```
                den = e.real * e.real + e.imag * e.imag;        /* absolute square      */
                power = pc[P] / den;                            /*Eq (46) in Chapter IV*/
                arpsdlog[q] = 10.0 * log10(power);              /* 10*log10             */
                q++ ;
        }
}
```

DSP.H

```
/* Header file for the function prototypes defined above                    */

void xcor(float *x,float *Rx,int N,int P);
void burg(float *x,float *bpc,int P,int NM1);
void levpre (float *ac,float *pc, int P,int size);
void levref (float *ac, float *rc, int PM1) ;
void rctopc (float *rcof,float *pc,int P,int size) ;
void schur(float *Rx,float *src,int P,int PM1) ;
void spschur (float *Rx, float *src, int P);
void spectra (float *pc,float *arpsdlog,int P, int W, float freqres) ;
```

APPENDIX B: TMS320C30 ASSEMBLY CODE FOR AUTOCORRELATION

This appendix lists the TMS320C30 assembly language code for the autocorrelation function. The object file of this code is executed faster than the object file obtained from the routine XCOR.C in Appendix A.

XCOR.ASM

```
*******************************************************
*   TMS320C30 C COMPILER    Version 4.50
*******************************************************
        .version        30
FP      .set            AR3
        .globl  _xcor
*******************************************************
* FUNCTION DEF : _xcor
*******************************************************
_xcor:
        PUSH FP
        LDI    SP,FP
        PUSH R4
        PUSH R5
        PUSHF R6
        PUSH AR4
        PUSH AR5

        LDI    *-FP(2),IR1          ; x  (input)
        LDI    *-FP(3),AR5          ; Rx (output)
        LDI    *-FP(4),BK           ; N  (data length)
```

```
        LDI    *-FP(5),R4          ; P  (lag size)

        FLOAT  BK,R0               ;
        CALL   INV_F30             ; 1 / N
        LDF    R0,R6               ;

        LDI    0,R3                ; m=0
        LDI    R4,AR6              ;
        SUBI   1,AR6               ;
LOOP                               ; for (m=0; m<(P+1))
        LDF    0.0,R2              ; z=0
        SUBI   R3,BK,R5            ; k=N-m
        LDI    IR1,AR2             ; x[n]
        ADDI   R3,AR2,AR4          ; x[n+m]
        SUBI   1,R5                ; k-1
        LDF    0.0,R0              ;

        RPTS   R5                  ; for (n=0; n<N-m)
        ADDF   R0,R2               ;
||      MPYF   *AR2++,*AR4++,R0    ;    z += x[n] * x[n+m]

        ADDF   R0,R2               ; end
        DBNZD  AR6,LOOP            ; end of LOOP
        MPYF   R6,R2,R0            ; z * 1/N
        STF    R0,*AR5++           ; Rx[m]
        ADDI   1,R3                ; m+1

        LDI    *-FP(1),R1
        LDI    *FP,FP
```

```
        POP   AR5
        POP   AR4
        POPF  R6
        BD    R1
        POP   R5
        POP   R4
        SUBI  2,SP

*********************************************************
* UNDEFINED REFERENCES                          *
*********************************************************
        .globl  INV_F30
        .end
```

APPENDIX C: BATCH FILES

All the batch files used in this work are listed in this appendix.

MKOPT.BAT

```
rem   This batch file optimizes the AR routine codes listed in Appendix A, and then obtains
rem   the executable object files.

cl30 -o -s -al xcor.c burg.c levpre.c levref.c  rctopc.c schur.c spschur.c spectra.c
```

MKLIB.BAT

```
rem   This batch file collects the optimized, executable object files obtained by the
rem   MKOPT.BAT batch file in a single library file.  The name of the library is dsplib.lib.

ar30 -r dsplib.lib xcor.obj burg.obj levref.obj levpre.obj rctopc.obj schur.obj spschur.obj spectra.obj
```

MKXCOR1.BAT

```
rem   This batch file is used to show the data transfer between the C30 board and the PC
rem   and to check the results of the functions. This is written specially for the programs
rem   described in Appendix D, and to check the functional correctness of the routine
rem   XCOR.C. The first line compiles and links the C30 code with the DSPLIB.LIB library
rem   file and runtime  support library RTS30.LIB. The second line compiles and links the
rem   PC C code with BCC and LSI PC libraries in the C30 directory. (Here, C30 is the
rem   name of the directory that holds the library files)

cl30 -s -al txcor.c -z -cr -m txcor.map lsicmap.cmd lsiboot.obj xcor.obj -l rts30.lib -o txcor.out
bcc -ml -P-CP -Ic:\c30;c:\cpp\lib -ic:\c30;c:\cpp\include -erxcor rxcor.cp lm30dev.lib
```

MKXCOR2.BAT

```
rem   This batch file is used to obtain the executable object file of the functions written in
rem   TMS320C30 assembly code, to show the data transfer between the C30 board and
rem   the PC, and to check the results of the functions. This is written specially for the
rem   programs described in Appendix D, and to check the functional correctness of the
rem   routine XCOR.ASM listed in Appendix B. The first line obtains the executable object
rem   file of the assembly language code XCOR.ASM with the name xcor.obj. The name
rem   of the listing file is xcor.lst. The second line compiles and links the C30 code with the
rem   runtime support library RTS30.LIB. The third line compiles and links the PC C code
rem   with BCC and LSI PC libraries in the C30 directory. (Here, C30 is the name of the
rem   directory that holds the library files). Note that in the second line, xcor.obj is used
rem   before the -l option. Because, we did not create the library dsplib.lib from this
rem   particular xcor.asm subroutine, which is actually faster than the one used in dsplib.lib.

asm30 xcor.asm xcor.obj xcor.lst
cl30 -s -al txcor.c -z -cr -m txcor.map lsicmap.cmd lsiboot.obj xcor.obj -l rts30.lib -o txcor.out
bcc -ml -P-CP -lc:\c30;c:\cpp\lib -ic:\c30;c:\cpp\include -erxcor rxcor.cp lm30dev.lib
```

MKINT_EX.BAT

```
rem   This batch file is used to show the interrupt service routine example. This is
rem   written specially for the programs described in Appendix E. The first line compiles and
rem   links the C30 code with the runtime support library RTS30.LIB. The second line
rem   compiles and links the PC C code with BCC and LSI PC libraries in the C30 directory.
rem   (Here, C30 is the name of the directory that holds the library files).

cl30 -s -al int_ex.c -z -cr -m int_ex.map lsicmap.cmd lsiboot.obj -l rts30.lib -o int_ex.out
bcc -ml -P-CP -LC:\c30;c:\cpp\lib -IC:\c30;c:\cpp\include -erint_ex rint_ex.cp lm30dev.lib
```

APPENDIX D: DATA TRANSFER BETWEEN PC AND C30 BOARD

The following two programs illustrate the data transfer between the PC and the C30 board as described in Chapter III. The first program, *30XCOR.C*, is the C30 program, and the second program, *RXCOR.CP*, is the PC program. The PC program loads the input data into the C30 memory, the autocorrelation of the data is computed by the C30, and the result is sent back to the PC for display. *MKXCOR1.BAT* in Appendix C is the batch file to invoke the compiler and linker with necessary options. In order to run the program, type *MKXCOR1* to compile and link the program, then type *RXCOR* to execute the program.All the files should be in the current directory. The results are displayed on the PC screen.

30XCOR.C

```
/* This is a C program for the C30 board. The program finds the autocorrelation  coefficients
of the data passed down form the PC and passes the  result back to the PC. It works in
conjunction with the PC program RXCOR.EXE                                          */

#include <c:\tms_tool\math.h>
#include "dsp.h"                          /* header file for the function      */

#define N   160                           /* data length                       */
#define P    10                           /* autocorrelation lag size          */
#define PP1 P+1
#define PP3 P+3

float x[N];                               /* input data from the PC            */
float ac[PP3];                            /* output result to the PC           */

long  *PCproceedflag, *DSPproceedflag;    /* PC communication flags            */
extern long  Comm0, Comm1;                /*location of the communicationflags */
extern float  *Comm2, *Comm3;             /* pointers to the locations in dual  */
                                          /* access memory area for the input  */
                                          /* and output                        */
long *t0pr, *t0gc, *t0cr;                 /* pointers to the Timer 0 registers */
```

```
main()
{

asm("     OR    800h, ST");          /* Enables the cache                 */

PCproceedflag = &Comm0;            /* Assign meaningful names to        */
DSPproceedflag = &Comm1;           /* pointers to flags used to         */
                                   /*  communicate with PC              */
Comm2 = x;                         /* Put starting address of input array */
                                   /* in absolute location so the PC can */
                                   /* find the  arrays                  */
*PCproceedflag = 1;                /* tell pc address is ready          */

while (*DSPproceedflag == 0);      /* Wait for PC to download input     */
                                   /*  array                            */
*DSPproceedflag = 0;

t0gc = (long *) 0x808020;          /* Initialize pointers to Timer 0    */
t0pr = (long *) 0x808028;          /*  registers: global control register, */
t0cr = (long *) 0x808024;          /* period register, and counter register*/

*t0gc = 0;                         /* Timer 0 control register resets the */
                                   /*  timer                            */
*t0pr = 0X05B8D8;                  /* period register is set to a value big */
                                   /* enough to complete the execution  */
*t0gc = 0x0002C1;                  /* Start the timer.                  */

ac[PP1] = *t0cr;                   /* counter is read before the function */
                                   /*  starts                           */
xcor(x,ac,N,P);                    /* function call                     */

ac[P+2] = *t0cr;                   /* counter is read after the function is*/
                                   /* complete                          */
/* The difference of the value in the counter register before and after the function call is the
execution time. Note that counter is incremented once in every 120 nsec.          */

Comm3=ac;                          /* Put starting address of output array*/
                                   /* in absolute location so the PC can */
                                   /* find the array                    */
*PCproceedflag = 1;                /* tell pc data ready                */
while (*DSPproceedflag == 0);      /* Wait for PC to download an array  */
*DSPproceedflag = 0;
}
```

RXCOR.CP

```
/* This is a PC program that downloads and runs 30XCOR.OUT on the C30 board.*/

#include "tms30.h"                      /* header file for the interface library */
#include <stdio.h>                      /* runtime support header files          */
#include <stdlib.h>
#include "mucoprt.cp"                   /* PC function that prints an array and*/
                                        /* a matrix                              */

#define BOARDADR    0x290               /* Factory default I/O address.          */

#define COMM0           0x30000         /* Start of dual access memory area      */
#define PCPROCEEDFLAG   COMM0+0         /* addresses between the PC and the      */
#define DSPPROCEEDFLAG COMM0+0          /* C30. These addresses are defined      */
#define INPUT           COMM0 + 2       /*  in LSICMAP.CMD  map file.            */
#define OUTPUT          COMM0 + 3

#define MAXCOUNT    200000              /* Max. delay while waiting  for the     */
                                        /* C30 to response                       */
#define N     160                       /* data length                           */
#define P     10                        /* autocorrelation lag size              */
#define PP3   P+3

void main(void)
{

unsigned short   loadstat;
long a;
unsigned long  inloc,outloc;
float x[N];                             /* input data                            */
float ac[PP3];                          /* autocorrelation lag values from       */
                                        /* R(0) to R(P), and the value in the    */
                                        /* Timer 1 counter register before       */
                                        /* and after the function call           */
FILE *fpi,*fpo;
int i;

/* open file to read input data; DATA160.IN holds the input data                 */

if (( fpi = fopen ("DATA160.IN","r")) == NULL) {
```

```
        printf ( "Unable to open input data file \n");
        exit(0);
}

/* open file to read Correlation Coefficients; COR.IN will hold the results        */

if (( fpo = fopen ("COR.IN","w")) == NULL) {
        printf ( "Unable to open output file \n");
        exit(0);
}

/* read data into x array */

for (i=0; i<N; i++) {
        if (!feof(fpi))
                fscanf(fpi,"%f",&x[i]);
        else {
                printf("End of input file before all data read\n");
                exit(0);
        }
}
fclose(fpi);

printf ("Successfully loaded DATA160.IN\n");

/* Initialize board:  */

SelectBoard(BOARDADR);
loadstat = coffLoad("30xcor.out");              /* Special load function; required*/
                                                /* with -cr (RAM) linker option */
if (loadstat != 0) {
        printf("\n\nError During Program Load!!!!\n");
        printf("coffLoad() returned %x\n\n", loadstat);
        exit (0);
}

Put32Bit(PCPROCEEDFLAG,DUAL,0x0L);      /* Make sure flag is set to zero*/
Put32Bit(DSPPROCEEDFLAG,DUAL,0x0L);     /* Make sure flag is set to zero. */
Reset();                                /* Start the DSP program running. */

/* Wait till DSP has pointers to output arrays ready */

for (a=0; a<MAXCOUNT && (Get32Bit(PCPROCEEDFLAG,DUAL) != 0x1L); a++);
```

```
if (a==MAXCOUNT) {
        printf("Timeout waiting for array pointers!!!\n");
        exit(0);
}

Put32Bit(PCPROCEEDFLAG,DUAL,0x0L);

inloc=Get32Bit(INPUT,DUAL);

WrBlkFlt(inloc, DUAL, N, x);                    /* Download the x vector.     */

Put32Bit(DSPPROCEEDFLAG,DUAL,0x1L);     /* Tell DSP to start.             */

/* Wait for DSP to finish calculation of XCOR then upload. */

for (a=0; a<MAXCOUNT && (Get32Bit(PCPROCEEDFLAG,DUAL) != 0x1L); a++);
                /* Wait loop -- till answer array is ready. */
if (a==MAXCOUNT) {
        printf("Timeout waiting for xcor to be calculated!!!\n");
        exit(0);
}

Put32Bit(PCPROCEEDFLAG,DUAL,0x0L);

outloc = Get32Bit(OUTPUT,DUAL);                 /* Get actual starting addresses ofarray*/

RdBlkFlt(outloc,DUAL,PP3,ac);                   /* Get result of output       */

printf("autocorrelation");
prtarray(ac,PP3);                               /* prints the result          */

for (i=0; i<PP3; i++)                           /* saves the result in a file */
        fprintf(fpo,"%g\n",ac[i]);
fclose(fpo);
}
```

APPENDIX E: INTERRUPT SERVICE ROUTINE EXAMPLE

The following two programs illustrate the use of interrupt service routines described in Chapter III. The first program, *INT_EX.C*, is the C30 program, and the second program, *RINT_EX.CP*, is the PC program that runs the "*INT_EX.OUT*", the output file of the C30 program. *MKINT_EX.BAT* in Appendix C is the batch file to invoke the compiler and linker in a single step. In order to run the program, first connect the channels A and B on the board to the channels A and B on the oscilloscope. Then, type *MKINT_EX* to compile and link the program, and then type *RINT_EX* to execute the program. Two sinusoid signals with different amplitudes will be seen on the oscilloscope screen.

INT_EX.C

```
/* The C program for the C30 board INT_EX.C echos the samples on the channel A of ADC
onto  the channel A of DAC and multiplies the signal on the channel A of ADC by three and
echos onto the channel B of DAC.                                                    */

#include <c:\tms_tool\math.h>
#define  ADC_A  0x804000              /* Memory locations of the channels  */
#define  DAC_A  0x804000              /* A and B of the C30 board          */
#define  DAC_B  0x804001

long *t1cr, *t1pr, *t1gc;

main()
{

asm("   OR   800h, ST");             /* Enables the cache                 */

t1gc = (long *) 0x808030;            /* Timer 1 Global Control Register   */
t1pr = (long *) 0x808038;            /* Timer 1 Period Register           */

*t1gc = 0x601;
*t1pr = 0x3F9;                       /* Sampling frequency is 8192 Hz     */
*t1gc = 0x6c1;                       /* start the Timer 1                 */
```

```
asm("  OR  2h, IE");                    /* set up the INT1 bit (second bit) in */
                                        /* the IE register                     */
asm("  OR  2000h, ST");                 /* set up the global interrupt enable  */
                                        /* (13th bit) bit in the ST register   */
while(1);                               /* loop forever                        */
}

asm (" .sect   \".int02\"");            /* Locate next statement in vector table. */
asm (" .word  _c_int02 ");              /* Inserts vector to interrupt routine.   */
asm ("  .text          ");              /* Put rest of the code in ".text" section */

/*****************************************************************
INTERRUPT SERVICE ROUTINE:
*****************************************************************/

c_int02 ()                              /* A/D & D/A end-of-convert interrupt. (INT1) */
{

int sample;

sample = *(int *) ADC_A;                /* Read the current sample .           */
*(int *) DAC_B = sample*3;              /* Output value*3 to D/A B.            */
*(int *) DAC_A = sample;                /* Output value to D/A A.              */
}
```

2. RINT_EX.CP

```
/* This is a PC program that downloads and runs INT_EX.OUT on the C30 board. */

#include "tms30.h"                      /* Header file for the interface library */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{

int  loadstat;
```

```
/* Initialize board:   */
SelectBoard(0x290);
loadstat = coffLoad("int_ex.out");     /* Special load function; required with -cr option*/

/* Start the C30 program.    */

Reset();

printf("C30 program (int_ex.out) is running.\n");
printf("You should see an echo of ADC A on DAC A.\n");
printf("You should see (ADC A)*3 on DAC B.\n");
}
```

APPENDIX F: LSICMAP.CMD MEMORY MAP FILE

```
/* LSICMAP.CMD: Memory map file for LSI TMS320C30 System Board, for use with the
C Compiler.                                                              */

/* All memory is RAM.        */

MEMORY          /* This maps memory SECTIONS to the board hardware.    */
{

/* EXTERNAL SRAM ON THE MAIN BOARD:                                      */

/* Locations 0 to C0h are reserved for interrupt vectors and Debug Monitor usage.*/
/* Although you could start using memory at C1h, this map starts at 100h -- to    */
/* allow for future Monitor expansion, and for ease of adding hex addresses offsets*/

   VECTS: origin=000000h  length=00000ch      /* Interrupt vectors.          */
   BANK0: origin=000100h  length=00ff00h      /* Standard SRAM (0-wait).     */
   BANK1: origin=010000h  length=010000h      /* SRAM upgrade option.        */
   BANK2: origin=020000h  length=010000h      /* SRAM upgrade option.        */
   BANK3: origin=030000h  length=00f400h      /* Std. dual access (1-wait)   */

/*  Bank 3 is dual-access between the C30 and the PC. The length shown is for the default
64Kx4 devices, but 16Kx4 can be used.  In both cases, the top c00h locations are reserved
for Debug Monitor use.  If you will never use the debug monitor, your programs can use this
area.                                                                    */
```

```
/* CACHED DRAM MEMORY EXPANSION ON THE DAUGHTER BOARD: */

   EXPAND: origin=400000h  length=400000h   /* One of various options.        */

/*  ON-CHIP MEMORY:                                                           */

   BLOCK0: origin=809800h  length=0000400h
   BLOCK1: origin=809c00h  length=0000400h
}

SECTIONS    /*  Assigns program sections to the MEMORY statement, above.    */
            /*  The .data section, below, is not used by the linker to link C  */
            /*  C compiler output files.  It is used  by the linker when it is  */
            /*  linking Assembler output files. The section is included in this  */
            /*  "map" file so that the same map can be used to link files  */
            /*  produced by either the Assembler or C Compiler  */
            /* (useful if you write some functions in assembly language and  */
            /* happen to use the  .data section).  */
{
      .text:          {}      >BANK0
      .bss:
      {
        _Comm0 = .;  . += 1;        /* Define global address labels that can   */
        _Comm1 = .;  . += 1;        /* be used for communication between the   */
        _Comm2 = .;  . += 1;        /* PC and DSP programs.  These locations   */
        _Comm3 = .;  . += 1;        /* will each occupy one 32-bit word        */
        _Comm4 = .;  . += 1;        /* starting at zero offset from the        */
        _Comm5 = .;  . += 1;        /* beginning of the .bss section.          */
        _Comm6 = .;  . += 1;
```

```
    _Comm7 = .;  . += 1;        /* If you need more locations,                */
    _Comm8 = .;  . += 1;        /* you could add more "Comm" locations        */
    _Comm9 = .;  . += 1;        /* or you could create a "hole" in memory     */
    _Comm10 = .;  . += 1;       /* here that you address using absolute       */
    _Comm11 = .;  . += 1;       /* pointers (instead of these labels).        */
    _Comm12 = .;  . += 1;
    _Comm13 = .;  . += 1;
    _Comm14 = .;  . += 1;
    _Comm15 = .;  . += 1;

  } >BANK3

  .data:        {}    >BANK3
  .cinit:       {}    >BANK3
  .stack:       {}    >BLOCK0

/* .sysmem:     {}    >BANK2*/

/* Forces Reset and Interrupt Vectors to absolute locations:Your C source code   */
/* should initialize these locations using "ASM" inline assembly macros (except for */
/* location 00, which is initialized in the LSIBOOT.OBJ startup file.            */

  .int00  00h: {}       /*  Reset (Power-on or otherwise).                      */
  .int01  01h: {}       /*  INT0                                                */
  .int02  02h: {}       /*  INT1 (A/D & D/A end of convert).                    */
  .int03  03h: {}       /*  INT2                                                */
  .int04  04h: {}       /*  INT3                                                */
  .int05  05h: {}       /*  XINT0                                               */
  .int06  06h: {}       /*  RINT0                                               */
```

```
        .int07  07h: {}         /* XINT1                                */
        .int08  08h: {}         /* RINT1                                */
        .int09  09h: {}         /* TINT0                                */
        .int10  0ah: {}         /* TINT1                                */
        .int11  0bh: {}         /* DINT                                 */
}

-heap 4096      /*Sets the size of the .sysmem section to 4K words. The default is 1K. */
```

LIST OF REFERENCES

1. *TMS320C30, User's Guide*, Texas Instruments, Inc., Dallas, TX, 1992.

2. *TMS320 Floating Point DSP Assembly Language Tools, User's Guide*, Texas Instruments, Inc., Dallas, TX, 1992.

3. *TMS320 Floating Point DSP Optimizing C Compiler, User's Guide*, Texas Instruments, Inc., Dallas, TX, 1992.

4. Rulph Chassaing, *Digital Signal Processing with TMS320C30*, Wiley-Interscience, New York, NY, 1992.

5. M. Shields, EC 4930 Digital Signal Processing Hardware Course Notes (1995), Naval Postgraduate School, Monterey, CA.

6. *TMS320C30 Spectrum System Board User's Guide*, Spectrum Signal Processing, Inc. Blaine, WA, 1990.

7. S. Kay, *Modern Spectral Estimation*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

8. J. P. Burg, "A New Analysis Technique for Time Series Data," NATO Advanced Study Institute on Signal Processing with Emphasis on Underwater Acoustics, 1968. D. Chielders, ed., *Modern Spectrum Analysis*, IEEE Press, 1978.

9. E. Robinson and S. Treitel, "Maximum Entropy and the Relationship of the Partial Autocorrelation to the Reflection Coefficients of a Layered System," *IEEE Trans. Acoust., Speech, Signal Process.*, ASSP-28, 22(1980).

10. C. W. Therrien, *Discrete Random Signals and Statistical Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

11. B. Madhukar, V. K. Anuradha, A. W. Ubale, "AR Parameter Estimation Routines for The Motorola DSP56100 Family," *Proceedings of The International Conference on Signal Processing Applications and Technology (ICSPAT)*, Vol-I, 1994.

INITIAL DISTRIBUTION LIST

		No.Copies
1.	Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944 Ft. Belvoir, VA 22060-6218	2
2.	Library Code 013 Naval Postgraduate School Monterey, CA 93943-5101	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
4.	Professor M. Shields, Code EC/Sl Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
5.	Professor M. Tummala, Code EC/Tu Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
6.	Professor H. H. Loomis, Jr., Code EC/Lm Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
7.	Deniz Kuvvetleri Komutanlığı Personel Daire Başkanlığı Bakanlıklar, Ankara, 06600 TURKEY	2
8.	Mücahit Karasu Güleser sok. No:4 Çeliktepe, İstanbul, 80650 TURKEY	2